FlatDD: Parallel Quantum Circuit Simulation using Decision Diagram and Flat Array

SHUI JIANG, The Chinese University of Hong Kong, China
HENGRUI CHEN, Zhejiang University, China
RONGLIANG FU, The Chinese University of Hong Kong, China
LUKAS BURGHOLZER, Technical University of Munich, Germany and Munich Quantum Software Company, Germany
ROBERT WILLE, Technical University of Munich, Germany and Munich Quantum Software Company, Germany
TSUNG-YI HO, The Chinese University of Hong Kong, China
TSUNG-WEI HUANG, University of Wisconsin-Madison, USA

Quantum circuit simulator (QCS) is essential for designing quantum algorithms because it assists researchers in understanding how quantum operations work without access to expensive quantum computers. Traditional array-based QCSs suffer from exponential time and memory complexities. To address this problem, Decision Diagram (DD) was introduced to compress simulation data by exploring the circuit regularity. However, for irregular circuit structures, DD-based simulation incurs significant runtime and memory overhead. To overcome this challenge, we present FlatDD, a parallel QCS that capitalizes on the strength of both DD- and array-based approaches. FlatDD parallelizes the simulation workload at multiple levels and leverages caching to reuse historical results. To further enhance the simulation performance for deep circuits, FlatDD introduces a gate-fusion algorithm to reduce the computational cost. Compared to state-of-the-art QCSs on commonly used quantum circuits, FlatDD achieves 54.35× speed-up and 1.91× memory reduction. [1]

ACM Reference Format:
Shui Jiang, Hengrui Chen, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. FlatDD: Parallel Quantum Circuit Simulation using Decision Diagram and Flat Array. 1, 1 (July 2026), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Quantum computing (QC) has the potential to efficiently handle certain types of problems that are classically intractable, such as quantum chemistry [10], finance [38], and cryptography [23]. Powered by two fundamental quantum phenomena, *superposition* and *entanglement*, many available quantum computers have shown significant speed-up over classical

[1]Preliminary version of this paper was originally published in the International Conference on Parallel Processing (ICPP) [66].

Authors' Contact Information: Shui Jiang, sjiang22@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR, China; Hengrui Chen, hengruichen@zju.edu.cn, Zhejiang University, Hangzhou, China; Rongliang Fu, The Chinese University of Hong Kong, Hong Kong SAR, China, rlfu@cse.cuhk.edu.hk; Lukas Burgholzer, lukas.burgholzer@tum.de, Technical University of Munich, Munich, Germany and Munich Quantum Software Company, Munich, Germany; Robert Wille, robert.wille@tum.de, Technical University of Munich, Munich, Germany and Munich Quantum Software Company, Munich, Germany; Tsung-Yi Ho, tyho@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong SAR, China; Tsung-Wei Huang, tsung-wei.huang@cse.cuhk.edu.hk, University of Wisconsin-Madison, Madison, Wisconsin, USA.

Manuscript submitted to ACM

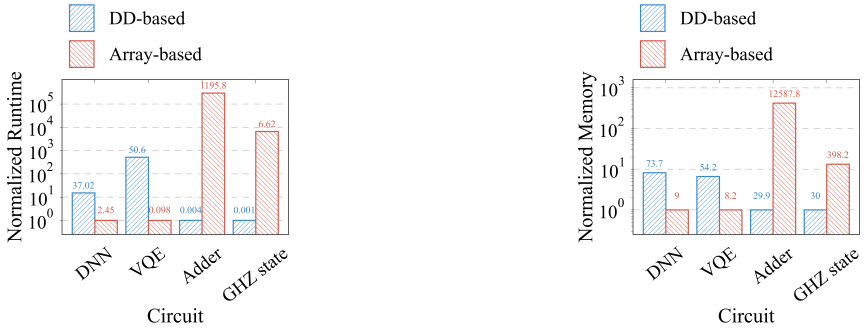


Fig. 1. Normalized runtime and memory results between a decision diagram (DD)-based simulator [105] and an array-based simulator [22] on four structurally different quantum circuits. Bar heights are normalized within each circuit, while the label above each bar reports the absolute value (seconds for runtime and MB for memory). DD can compress simulation data in a compact graph-based data structure by exploring regularity in the circuit, such as state amplitude distribution and gate matrices structures.

computers. For example, a recent photonic quantum computer `Jiuzhang` can solve the Gaussian boson sampling problem within five minutes, whereas a supercomputer needs billions of years [101]. To enable widespread use of this quantum advantage, researchers have been actively building software stacks to support quantum computer designs [4, 13, 44, 65, 88, 96].

Among various QC supporting tools, developing an efficient *quantum circuit simulator* (QCS) on a classical computer is a crucial task because it helps researchers understand how quantum operations work and verify the functionality of a quantum algorithm. However, this task is extremely challenging because it demands large space and time complexity to compute state amplitudes of qubits. For instance, state-of-the-art QCSs [1, 4, 5, 22, 69, 75, 95, 97, 100] use arrays to represent quantum gate matrices and state vectors. An n-qubit circuit may result in a worst case of multiplying a $2^n \times 2^n$ quantum gate matrix by a $2^n \times 1$ state vector. To tackle this challenge, [92, 105] have proposed *decision diagram* (DD) to compress simulation data in a compact graph-based data structure by exploring regularity in the circuit, such as state amplitude distribution and gate matrices structures. As a result, DD can significantly reduce the space complexity and largely improve the simulation time. It is widely used by many quantum software projects [13, 24, 25, 39, 40, 103].

Despite superior performance on regular circuit structures (e.g., quantum arithmetic), DD-based simulators cannot efficiently handle circuits that exhibit irregular distributions of state amplitudes, such as deep neural network (DNN) [11], variational quantum eigensolver (VQE), and Google's quantum supremacy circuits [8]. When simulating these irregular circuits, DD has few advantages but suffers from exponential runtime and memory overhead due to its graph structure–which would otherwise be more efficient using 1D arrays. Figure 1 illustrates this problem by showing the runtime and memory results between a DD-based simulator [105] and an array-based simulator [22] on two regular (Adder, GHZ State [94]) and two irregular (DNN, VQE) quantum circuits.

To solve this problem, we have identified an important property: In DD-based simulation, the quantum gate matrix has a regular structure as it can be recursively decomposed to unitary operators through Kronecker product. On the other hand, quantum state vectors exhibit a different behavior because of superposition and amplification over the state space [24]. Typically, the vector starts with a highly regular distribution and gradually becomes irregular as the simulation progresses. With this property, we present a parallel QCS called FlatDD that capitalizes on the strength of both DD- and array-based approaches. For clarity, hereafter we refer to the original FlatDD from [66] as FlatDD-v1, and to the enhanced version presented in this journal submission, which incorporates the proposed optimizations, as

FlatDD-v2. When referring to features common to both versions, we simply use "FlatDD". We summarize our technical contributions below:

- We introduce a hybrid data structure *DMAV*, which uses DD-based gate matrix and array-based state vector for matrix-vector multiplication. DD-based gate matrix enhances indexing efficiency, while array-based state vector avoids exponential overhead from irregularity.
- We introduce parallel DMAV algorithms that reuse simulation data through caching. Our DMAV algorithms overcome the limitation of DD-based simulation which is inherently sequential, thus largely enhancing its runtime scalability on a multicore architecture. **In FlatDD-v2**, we further introduce an optimized multi-threaded column-space DMAV algorithm, which further mitigates the memory and runtime overhead of caching, significantly improving the performance of DMAV simulation.
- We introduce a moving average-based algorithm to effectively decide when to convert the simulation from DD to DMAV. Since such a conversion can be time-consuming for large circuits, we introduce parallel algorithms to maximize the conversion efficiency. **In FlatDD-v2**, we further introduce a new task-parallel DD-to-array conversion algorithm to ensure load balancing across threads, significantly improving the runtime performance of DD-to-array conversion.
- We introduce a DMAV-aware gate-fusion algorithm to enhance FlatDD's efficiency in handling large quantum circuits with deep simulation length. **In FlatDD-v2**, we further incorporate a look-ahead mechanism in the gate-fusion algorithm to help prevent gate fusion from being trapped at a locally sub-optimal point.

We evaluated FlatDD on a set of widely-used quantum circuits from [8, 76, 94]. FlatDD can outperform two highly optimized state-of-the-art DD-based and array-based simulators, DDSIM [105] and Quantum++ [22], with significant runtime improvement. For example, FlatDD achieves an average of 54.35× and 19.44× speed-up over DDSIM [105] and Quantum++ [22]. The source code for this work is available at https://github.com/IDEA-CUHK/FlatDD.

2 Quantum Circuit Simulation

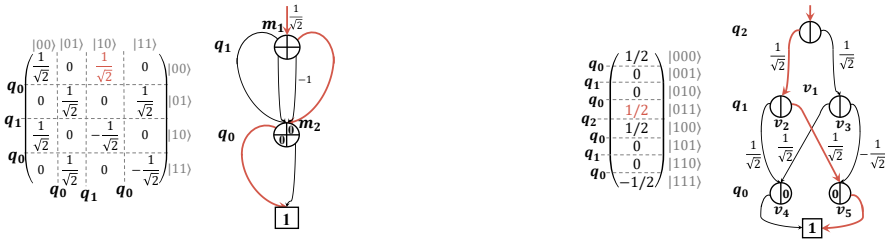
Given a quantum circuit, the goal of quantum circuit simulation is to derive the final state value after applying all quantum gate operations to an initial state. Mainstream simulation methods can be categorized to array-based and DD-based, explained below:

2.1 Array-based Simulation

In array-based simulation [1, 4, 5, 22, 69, 75, 95, 97, 100], gate matrices and state vectors are stored in 2D and 1D arrays. Multiplying a 2D array with a 1D array expresses the action of exerting a quantum gate on a state vector. For instance, when we apply a single-qubit Hadamard gate H to an input state $|\psi\rangle = |0\rangle$, it yields the resulting state $|\psi'\rangle$ (Equation 1).

$$|\psi'\rangle = H \cdot |\psi\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{1}$$

This can also extend to larger circuits with $n > 2$ qubits. However, we do not have to construct an entire $2^n \times 2^n$ gate matrix [95, 97]. Instead, array-based simulators can manipulate the amplitudes of state vectors locally. For instance, if we apply a single-qubit gate $U = (u_{ij})$ to the k-th qubit of state vector $|\psi\rangle = (a_i)^T$, the operation to derive resulting



(a) Matrix M in DD.

(b) Vector V in DD.

Fig. 2. Examples of array-based data represented in decision diagrams. Edges without labels have a weight of one by default.

state $|\psi'\rangle = (a'_i)^T$ can be expressed in Equation 2.

$$\begin{pmatrix} a'_{*\cdots*0_k*\cdots*} \\ a'_{*\cdots*1_k*\cdots*} \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \cdot \begin{pmatrix} a_{*\cdots*0_k*\cdots*} \\ a_{*\cdots*1_k*\cdots*} \end{pmatrix} \tag{2}$$

On the other hand, if we apply a two-qubit controlled gate $V = (v_{ij})$ to the state vector, where c represents the control qubit and t is the target qubit, the in-place manipulation of the state vector is expressed in Equation 3.

$$\begin{pmatrix} a'_{*0_c*\cdots*0_t*\cdots*} \\ a'_{*0_c*\cdots*1_t*\cdots*} \\ a'_{*1_c*\cdots*0_t*\cdots*} \\ a'_{*1_c*\cdots*1_t*\cdots*} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & v_{11} & v_{12} \\ 0 & 0 & v_{21} & v_{22} \end{pmatrix} \cdot \begin{pmatrix} a_{*0_c*\cdots*0_t*\cdots*} \\ a_{*0_c*\cdots*1_t*\cdots*} \\ a_{*1_c*\cdots*0_t*\cdots*} \\ a_{*1_c*\cdots*1_t*\cdots*} \end{pmatrix}. \tag{3}$$

2.2 Decision-diagram-based Simulation

Compared with array-based methods, decision diagram (DD)-based simulators [92, 104, 105] are particularly good at handling the regularity in gate matrices and state vectors. For example, if we equally partition the matrix in Figure 2a into four sub-matrices, we observe that the upper-left, upper-right, and lower-left sub-matrices are identical. The lower-right sub-matrix is exactly the opposite of the other three sub-matrices. As a result, these four sub-matrices can be efficiently stored as one single 2×2 sub-matrix with different weights. This idea extends to all gate matrices and state vectors, which can be recursively partitioned until they become scalar values. This process forms a graph-based data structure, DD, where nodes represent sub-matrices or sub-vectors at various partitioning levels connected by weighted edges. Figure 2 shows DD examples for gate matrix M and state vector V.

In Figure 2a, the root node m_1 represents the entire matrix M, and the four outgoing edges represent four equally partitioned sub-matrices in M. The partition runs on the most significant qubit, q_1, at the topmost level of the DD. Recursive partitions progress level by level, from the most to the least significant qubit. Each qubit matches a unique level in DD. The weight for m_1's incoming edge is $\frac{1}{\sqrt{2}}$, and the weights for its outgoing edges are $1, 1, 1$ and -1, corresponding to the upper-left, upper-right, lower-left, and lower-right sub-matrices. The weights are uniquely decided by *normalization* [92, 105]. After dividing m_1's incoming and outgoing weights, the four sub-matrices are identical, which can be expressed with one single node m_2. m_2 represents a 2×2 identity matrix, which can be further partitioned on qubit q_0. In this partition, the upper-left and lower-right elements point to terminal constant node *one*, and the upper-right and lower-left elements are zero. The matrix value of an index pair equals the product of edge weights along

the corresponding path in DD. For instance, for $M[0][2]$ (i.e., $M[|00\rangle][|10\rangle]$), we have $q_1 = q_0 = 0$ and $q_1 = 1, q_0 = 0$ for row and column indices, respectively. Multiplying the edge weights along the path (thick red edges in Figure 2a) yields $M[0][2] = \frac{1}{\sqrt{2}} \times 1 \times 1 = \frac{1}{\sqrt{2}}$.

In Figure 2b, the root node v_1 represents the entire state vector. We equally partition the vector into two sub-vectors on the most significant qubit q_2. The two sub-vectors are neither zero vectors nor a scalar multiple of each other. Therefore, they are represented in two unique nodes v_2 and v_3. The incoming weights for v_2 and v_3 are $\frac{1}{\sqrt{2}}$, also uniquely determined by normalization. Thus, the weight products along the paths to v_2 and v_3 are both $\frac{1}{\sqrt{2}}$. After dividing the weight products, the two sub-vectors represented by v_2 and v_3 can be further partitioned into $\begin{pmatrix}\frac{1}{\sqrt{2}} & 0\end{pmatrix}^T$, $\begin{pmatrix}0 & \frac{1}{\sqrt{2}}\end{pmatrix}^T$, $\begin{pmatrix}\frac{1}{\sqrt{2}} & 0\end{pmatrix}^T$ and $\begin{pmatrix}0 & -\frac{1}{\sqrt{2}}\end{pmatrix}^T$ on q_1 level, where the first and the third are identical, represented in node v_4, and the second and the fourth are opposite, represented in node v_5 with opposite incoming weights. After further dividing weight products, v_4 and v_5 represent $\begin{pmatrix}1 & 0\end{pmatrix}^T$ and $\begin{pmatrix}0 & 1\end{pmatrix}^T$ on the q_0 level, respectively. Similarly, the amplitude of an index is equal to the product of edge weights along the corresponding path in DD. For example, to determine $V[3]$ (i.e., $V[|011\rangle]$), we have $q_2 = 0$, $q_1 = 1$ and $q_0 = 1$. Multiplying the edge weights along the path (thick red edges in Figure 2b) gives $V[3] = 1 \times \frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} \times 1 = \frac{1}{2}$.

DD-based matrix-vector multiplication is done in a depth-first-search (DFS) fashion, where each matrix node always finds its corresponding vector node counterpart on the same level. Identical matrix-vector multiplications are avoided using hash tables [92, 104, 105].

3 Algorithm

In this section, we discuss the technical details of our FlatDD algorithm. Figure 3 gives an overview of FlatDD. As aforementioned, the quantum state vector typically begins with a highly regular distribution and gradually turns irregular as the simulation progresses. For example, in Figure 3, DD-based simulation has a fast runtime until about 24 gates at which the cost to maintain an irregular DD-based state vector outweighs its advantage. Considering this property, FlatDD begins with DD-based simulation using DDSIM [105] and converts to parallel simulation with DD-based gate matrix and array-based state vector multiplication (DMAV). We record a window of the DD size using moving average and examine if the state vector's regularity has experienced a significant change. When such a change happens, we use a parallel algorithm to convert the state vector from DD to array (Section 3.1), thereby converting the simulation from DD-based to DMAV. In Section 3.2, we introduce efficient multi-threaded DMAV algorithms to maximize performance. To further enhance the performance of FlatDD on large, deep circuits with thousands of gates, we introduce a DMAV-aware gate-fusion algorithm (Section 3.3). Finally, in Section 3.4, we summarize the difference between FlatDD-v1 and FlatDD-v2. Throughout this paper, we use t to represent the number of threads and n to represent the number of qubits.

3.1 Conversion from DD-based Simulation to DMAV

When the regularity of a DD-based state vector decreases to a certain level, DD-based simulation becomes less advantageous due to the cost of maintaining a highly irregular DD structure. To address this problem, a potential solution is to convert the DD-based state vector to an array representation when the DD size exceeds a certain threshold. The challenge is thus to decide when and how to perform this conversion. To this end, we decide the conversion timing via moving average (Section 3.1.1) and introduce parallel algorithms to efficiently convert DD to array (Sections 3.1.2 and 3.1.3).

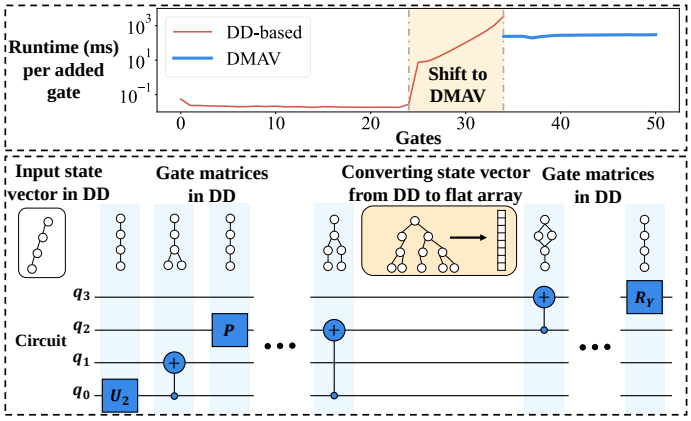


Fig. 3. Overview of the FlatDD algorithm. The top box displays the runtime for each quantum gate, while the bottom box shows the quantum circuit and simulation data structures used to compute state vectors.

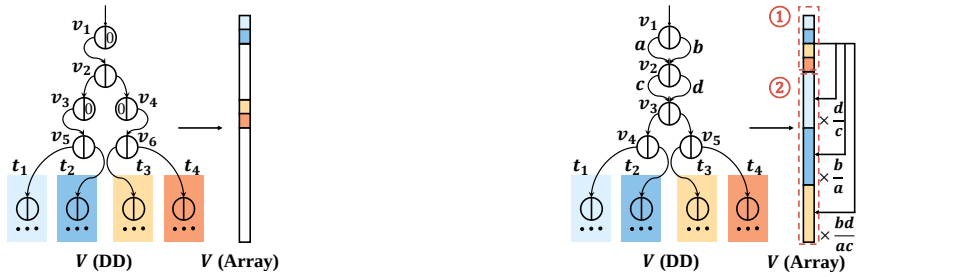
3.1.1 Conversion timing. To decide the best timing for conversion from DD to array, we monitor the DD size and perform conversion when the irregularity of DD experiences a drastic increase. Specifically, we apply a widely used learning method, *exponentially weighted moving average* [63] (EWMA). EWMA introduces little computational overhead and is suitable for different quantum circuits and simulation platforms. While simulating, gate i is assigned an EWMA value v_i based on the previous EWMA value v_{i-1} and the DD size s_i, of the state vector at gate i. v_i is computed through Equation 4.

$$v_i = \beta \cdot v_{i-1} + (1 - \beta) \cdot s_i \tag{4}$$

In Equation 4, β is a parameter and v_0 is initialized to 0. β captures the learning feature of EWMA based on a weighted window of historical data. To decide whether to convert at gate i, we compare $\epsilon \cdot v_i$ with s_i , where ϵ is a threshold parameter. (1) If $\epsilon \cdot v_i \geq s_i$, then s_i is not large enough to benefit from DMAV. (2) On the other hand, if $\epsilon \cdot v_i < s_i$, we convert from DD-based simulation to DMAV (Section 3.2) by converting state vector from DD to 1D flat array.

3.1.2 Multi-threaded Conversion from DD to Array. Although DD-based QCS DDSIM [105] already incorporates a conversion algorithm, it is inherently sequential. As a result, it can consume a large portion of the total simulation runtime. To overcome this challenge, in FlatDD-v1, we introduce a multi-threaded algorithm with two optimizations, zero skipping and scalar multiplication, to improve the conversion efficiency.

Figure 4 illustrates how our parallel conversion algorithm converts state vector V from DD to array using four threads (t_1, t_2, t_3, and t_4). In our algorithm, each DD node serves as a junction that divides threads across its two outgoing edges, progressing from top to bottom until further division of threads is not possible. Then, each thread traverses the unvisited nodes using DFS, and computes the state amplitudes through the products of weights along the traversal paths. In practice, however, edges may be zero. For example, as shown in Figure 4a, the right outgoing edges of v_1 and v_3, and the left outgoing edge of v_4 are zero. We address this problem with zero-skipping optimization: If a DD node's outgoing edge is zero, threads are not divided; instead, they all proceed along the non-zero edge. With this optimization, in Figure 4a, all four threads follow the left outgoing edge at node v_1 to node v_2. Here, the threads divide:



(a) Zero skipping: Avoiding idle threads at zeros.

(b) Scalar multiplication: Facilitating SIMD parallelism.

Fig. 4. Multi-threaded DD-to-array conversion algorithm with two optimizations. The four colors represent four threads.

two take left to v_3 and two take right to v_4. At nodes v_3 and v_4, the threads continue to nodes v_5 and v_6, respectively, and divide at v_5 and v_6. This optimization prevents any threads from remaining idle.

Additionally, we optimize the conversion by identifying opportunities for scalar multiplication, as it benefits from single instruction multiple data (SIMD) parallelism. For example, in Figure 4b, we can derive the second, third, and fourth quarters of the array from its first quarter with scalar multiplication. This is because v_1 and v_2 each have identical child nodes, indicating that the four quarters of the array are scalar multiples of one another [92]. Consequently, we break the conversion in Figure 4b into two steps: ① Convert the first quarter. With multi-threading, all four threads make two consecutive left turns at v_1 and v_2, for they each have identical child nodes, and divide at nodes v_3, v_4, and v_5, to convert the first quarter in parallel. ② SIMD-enabled scalar multiplication fills the second, third, and fourth quarters with three threads, using the first quarter data. For instance, the fourth quarter is calculated by multiplying the first quarter by scalar bd/ac. This optimization enhances performance by leveraging both SIMD and multi-threading parallelism.

3.1.3 Task-parallel Conversion from DD to Array. Although multi-threaded DD-to-array conversion is already highly efficient, it suffers from a workload imbalance challenge due to the uneven distribution of work among threads. For instance, different threads may convert sub-vectors of vastly different sizes due to the irregular structure of DDs. To address this challenge, in FlatDD-v2, we introduce a task-parallel conversion algorithm which describes the conversion of each sub-vector as a task and organizes these tasks into a task graph based on their dependencies. We delegate this task graph to a task scheduler [7, 12, 21, 55, 70], which effectively manages inter-task dependencies and load balancing across threads.

The task-parallel conversion algorithm constructs a task graph by traversing the DD using DFS from top to bottom, stopping at a threshold level L_C while skipping zero-weight edges. This traversal identifies two types of tasks: (1) conversion task, which sequentially converts DD-based sub-vector, and (2) scaling task, which applies scalar multiplication to array-based sub-vectors. Conversion tasks are applied to sub-vectors with root nodes at level L_C. Additionally, as demonstrated in Section 3.1.2, we optimize conversion by identifying opportunities for scalar multiplication, which benefits from SIMD parallelism. Specifically, if a DD node is pointed to by two edges, whether from the same parent DD node or from different parent DD nodes, one corresponding sub-vector can be generated first, and the other can then be obtained by scalar multiplication using the ratio of the two edge weights, thereby leveraging SIMD parallelism. Note that FlatDD-v1 only detects the special case in which the two edges originate from the same parent DD node, which misses many opportunities for SIMD-based scalar multiplication.

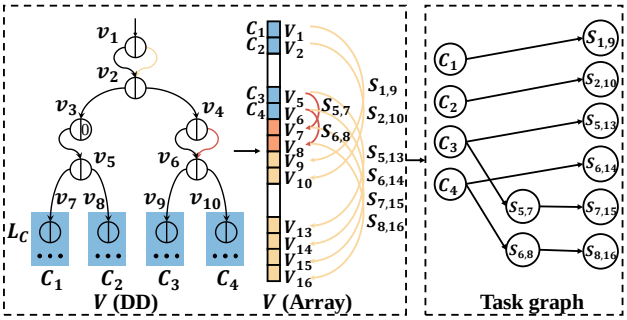


Fig. 5. Task-parallel DD-to-array conversion algorithm. We construct a task graph by traversing the DD using DFS.

In Figure 5, we first convert nodes v_7, v_8, v_9, and v_{10} into array-based sub-vectors V_1, V_2, V_5, and V_6, which correspond to conversion tasks C_1, C_2, C_3, and C_4 in the task graph. Since node v_4 has outgoing edges pointing to the same child node v_6, sub-vectors V_7 and V_8 are derived by scaling V_5 and V_6 using SIMD. The calculation of V_7 and V_8 is described by scaling tasks $S_{5,7}$ and $S_{6,8}$ in the task graph, which depend on C_3 and C_4, respectively. Similarly, since node v_1's outgoing edges point to the same child node v_2, sub-vectors V_9 through V_{16} are derived by scaling V_1 through V_8 using SIMD. This step is represented by scaling tasks $S_{1,9}$ through $S_{8,16}$, which depend on tasks C_1 through C_4, $S_{5,7}$ and $S_{6,8}$.

3.2 Simulation with DMAV

DMAV is different from existing matrix-vector multiplications [9, 91] because a DD-based gate matrix has a specific regularity property due to the tensor product. This regularity allows reusing computed results through caching. However, caching requires additional memory to store historical data, which can introduce overhead that, for certain gates, can outweigh its benefits. To effectively determine which gates benefit from caching, we introduce a computational cost model for DMAV. Using this model, we apply DMAV without caching when its computational cost is lower. Otherwise, we apply DMAV with caching.

This section first introduces single-threaded DMAV (ST-DMAV, Section 3.2.1). We then extend it to multi-threaded DMAV algorithms to enhance efficiency: multi-threaded row-space DMAV (MT-RS-DMAV, Section 3.2.2), which lacks computation reusability and does not support caching; and multi-threaded column-space DMAV (MT-CS-DMAV, Section 3.2.3), which supports caching but incurs runtime and memory overhead. To mitigate this overhead, we propose an optimized multi-threaded column-space DMAV (MT-CS-DMAV*, Section 3.2.4). Finally, we introduce a computational cost model in Section 3.2.5 to select the most efficient DMAV algorithm for each gate based on its cost.

3.2.1 Single-threaded DMAV. Algorithm 1 describes single-threaded DMAV (ST-DMAV) algorithm. The recursive function `ST_DMAV` traverses a DD-based gate matrix using DFS to multiply it by array-based state vector V and produce array-based state vector W. `ST_DMAV` recursively partitions the gate matrix into sub-matrices and state vectors V and W into sub-vectors until each reduces to a scalar value. A DD-based sub-matrix is located using its corresponding DD edge, while a sub-vector is located using its start index in V or W. `ST_DMAV` includes the following parameters: M_e is the DD edge corresponding to the current sub-matrix. V and W are the input and output array-based state vectors. L is the length of the current sub-vector in V and W, defaulting to 2^n (the full length of V and W). I_V and I_W are the start indices for the current sub-vectors in V and W, both of which defaulting to 0. f is the product of DD edge weights

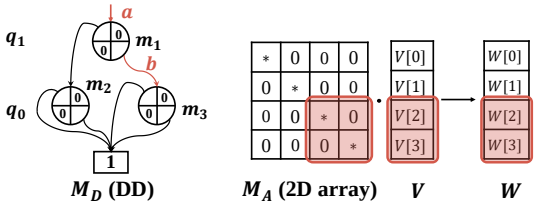


Fig. 6. Single-threaded DMAV. M_A is the 2D array representation of DD-based matrix M_D. The input state vector is V, and the resulting state vector is W.

along the current DD traversal path, initialized to M_e's weight by default (i.e., $M_e.w$). If each DD edge M_{e_i} has weight $M_{e_i}.w$ along a traversal path P, f is given by $f = \Pi_{i \in P} M_{e_i}.w$.

In Algorithm 1, if edge M_e is zero, we return immediately (line 2). At lines 3–5, if $M_e.n$ (the node pointed to by M_e) is not yet terminal, we traverse its four outgoing edges in row-major order (i.e., $M_e.n.e[i][j], \forall i, j \in \{0,1\}^2$) and call ST_DMAV recursively. We update the weight product with $f \cdot M_e.n.e[i][j].w$, I_V with $I_V + Lj/2$, and I_W with $I_W + Li/2$ while halving L. This recursive halving process continues until the sub-vectors reach a single scalar element, at which point $M_e.n$ also reaches a terminal node. In Figure 6, we use the lower-right edge of m_1 (i.e., $m_1.e[1][1]$) as an example. $m_1.e[1][1]$ corresponds to sub-matrix $M_A[2:4][2:4]$, where M_A is the 2D array representation of DD-based matrix M_D. $M_A[2:4][2:4]$ participates in multiplication $W[2:4] \leftarrow W[2:4] + M_A[2:4][2:4] \cdot V[2:4]$ (shaded in red). Since sub-vectors $V[2:4]$ and $W[2:4]$ each have a length of 2, their start indices are 2, and the weight product along the traversed DD path (marked in red) is ab, we call ST_DMAV($m_1.e[1][1], V, W, 2, 2, 2, ab$). Finally, if $M_e.n$ is a terminal node, we multiply f (which corresponds to $M_A[I_W][I_V]$ in 2D array, see Section 2.2) by $V[I_V]$, and add the result to $W[I_W]$ (line 6).

Overall, ST-DMAV outperforms array-based QCSs (e.g., Quantum++ [22]) in terms of state indexing. Unlike Quantum++, which requires $O(n)$ operations per state, ST-DMAV uses a recursive ST_DMAV function within a DD structure, enhancing data locality and reducing indexing to a constant average number of operations. This results in an $n\times$ increase in indexing speed for ST-DMAV compared to Quantum++.

Algorithm 1 Single-threaded DMAV (ST-DMAV)

Require: M_e, edge representing the current DD-based sub-matrix; V and W, array-based input and output state vectors; L, current sub-vector size; I_V and I_W, start indices of sub-vectors in V and W; f, weight product.

1: **function** ST_DMAV(M_e, V, W, $L = 2^n$, $I_V = 0$, $I_W = 0$, $f = M_e.w$)
2: **if** M_e **is** zero edge **then return**
3: **if** $M_e.n$ **is not** terminal node **then**
4: **for** $i, j \in \{0,1\}^2$
5: ST_DMAV($M_e.n.e[i][j]$, V, W, $\frac{L}{2}$, $I_V + \frac{L}{2}j$, $I_W + \frac{L}{2}i$, $f \cdot M_e.n.e[i][j].w$)
6: **else** $W[I_W] \leftarrow W[I_W] + f \cdot V[I_V]$; **return**
7: **end function**

3.2.2 Multi-threaded Row-space DMAV. We describe multi-threaded row-space DMAV (MT-RS-DMAV) using Algorithm 2, which consists of two functions: MT_RS_DMAV and RowAssign. The top-level function MT_RS_DMAV multiplies a DD-based gate matrix (represented by input parameter M_e, M's topmost DD edge) by array-based state vector V to produce the output state vector W. MT_RS_DMAV has two steps: first, it assigns multiplication tasks to t threads via

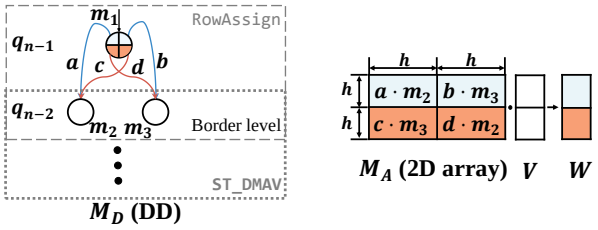


Fig. 7. Multi-threaded row-space DMAV on two threads, illustrated in blue and red colors. M_A is the 2D array representation of DD-based matrix M_D. The input state vector is V, and the resulting state vector is W.

recursive function RowAssign (line 4); second, it executes these tasks in parallel, with each thread running ST_DMAV (lines 5–7).

As shown in Figure 7, each thread evaluates M_A in row space by computing h rows from M_A and the entire vector V, resulting in an h-sized sub-vector in W, where $h = 2^n/t$. M_A is the 2D array representation of DD-based matrix M_D. RowAssign decomposes this computation for each thread into smaller multiplication tasks. It divides matrix M_A into $h \times h$-sized sub-matrices, and vectors V and W into h-sized sub-vectors. Each sub-matrix is paired with its corresponding sub-vectors to form a multiplication task. A sub-matrix in M_A is located using the corresponding DD edge in M_D, while a sub-vector can be located using its start index in V or W. We use 2D vectors v_M, v_V, and v_f (each of length t) to record the multiplication tasks for t threads. Here, v_M tracks the DD edges corresponding to sub-matrices, v_V records the start indices of sub-vectors in V, and v_f stores the weight products along DD traversal paths. We do not need to track the start indices of sub-vectors in W, as they are directly mapped to the thread index. Specifically, thread i produces $W[ih : (i+1)h]$.

RowAssign has the following parameters (line 10): M_e is the DD edge corresponding to the current sub-matrix. v_M, v_V, and v_f are the 2D vectors tracking the multiplication tasks. f is the product of DD edge weights along the current DD traversal path, defaulting to $M_e.w$. u is the current thread index for task assignment, defaulting to 0. I_V is the start index for the current sub-vector in V, defaulting to 0. l denotes the DD level of the current node, pointed to by edge M_e (i.e., $M_e.n$), initialized to the topmost level q_{n-1} (i.e., $l = n-1$).

In RowAssign, if M_e is zero, it returns (line 11). $n - \log_2 t - 1$ represents the *border level*: RowAssign ends here, and ST_DMAV starts from here. In Figure 7, the border level is q_{n-2} (i.e., $n - \log_2 t - 1 = n-2$). RowAssign spans levels q_{n-1} and q_{n-2} (i.e., $l = n-1, n-2$), while ST_DMAV spans levels $q_{n-2}, \cdots, q_0$. Upon reaching the border level, we push the sub-matrix's DD edge M_e, the sub-vector's start index I_V, and the weight product f of thread u to the corresponding vectors (lines 12–14). If the border level is still not reached, we traverse the four outgoing edges of $M_r.n$ in row-major order and recursively call RowAssign with the updated weight product $M_e.n.e[i][j].w \cdot f$, thread index $u + i \cdot t/2^{n-l}$ and sub-vector's start index, $I_V + 2^l j$, going one level lower to $l-1$ (lines 15–16). The purposes of $u + i \cdot t/2^{n-l}$ and $I_V + 2^l j$ are to split the group of t threads and V into halves, quarters, and so on, at each successive level, until all threads are allocated a multiplication task. After RowAssign, the blue thread in Figure 7 is assigned $a \cdot m_2 \cdot V[0:h]$ and $b \cdot m_3 \cdot V[h:2h]$, while the red thread is assigned $c \cdot m_3 \cdot V[0:h]$ and $d \cdot m_2 \cdot V[h:2h]$.

3.2.3 Multi-threaded Column-space DMAV. In column-space DMAV, we can reuse computed results through caching, reducing DMAV computation. For example, when evaluating the gate matrix M_A in Figure 8 in column space, focusing on its left two columns, two multiplication tasks occur: ① $am_2 \cdot V[0:2]$ and ② $bm_2 \cdot V[0:2]$. Without caching, both require four multiply-accumulate (MAC) [2] operations. However, ① and ② share identical left and right operands

Algorithm 2 Multi-threaded row-space DMAV (MT-RS-DMAV)

Require: M_e, edge representing a DD-based gate matrix; V and W, array-based input and output state vectors.

function MT_RS_DMAV(M_e, V, W)
  $h \leftarrow 2^n/t$
  $v_M = v_V = v_f \leftarrow \underbrace{\{\{\}, \cdots, \{\}\}}_{t}$
  RowAssign(M_e, v_M, v_V, v_f)
  parallel for $i \in [0, t)$
    for $j \in [0, size(v_M[i]))$
      ST_DMAV($v_M[i][j]$, V, W, h, $v_V[i][j]$, ih, $v_f[i][j]$)
end function

function RowAssign(M_e, v_M, v_V, v_f, $f = M_e.w$, $u = 0$, $I_V = 0$, $l = n-1$)
  if M_e **is** zero edge **then return**
  if $l == n - \log_2 t - 1$ **then**
    $v_M[u]$.push(M_e); $v_V[u]$.push(I_V);
    $v_f[u]$.push(f); **return**
  else for $i, j \in \{0,1\}^2$
    RowAssign($M_e.n.e[i][j]$, v_M, v_V, v_f, $f \cdot M_e.n.e[i]\ [j].w$, $u + \frac{i \cdot t}{2^{n-l}}$, $I_V + 2^l j$, $l-1$)
end function

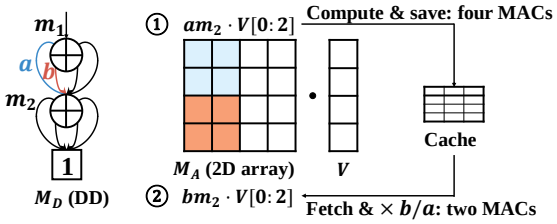


Fig. 8. Saving the number of MAC operations by reusing historical results in DMAV.

(sub-matrix DD node and sub-vector) and differ only in their scalar coefficients (a or b). Caching the result of ① allows its reuse for ② by scaling it by b/a, yielding only two MAC operations, compared to four without caching. Note that historical results can be reused only when each thread evaluates the gate matrix in column space instead of row space. This is because the right operand (sub-vector in V) remains unchanged for each thread, while the left operand (sub-matrix DD node) may have been computed in a previous step.

To enable caching in multi-threaded DMAV, we introduce multi-threaded column-space DMAV (MT-CS-DMAV) in Algorithm 3. In Algorithm 3, we present two functions. The top-level function MT_CS_DMAV performs DMAV in column space by multiplying a DD-based gate matrix (represented by the topmost edge M_e) by the array-based vector V to produce the array-based vector W. The function ColAssign assigns multiplication tasks to t threads. For previously computed multiplication tasks, we can fetch the results from the cache, while other multiplication tasks are computed using ST_DMAV (see Algorithm 1). We only apply caching to the multiplication tasks directly assigned by ColAssign to avoid frequent memory allocation and deallocation on different threads.

As shown in Figure 9, each thread computes h columns in M and an h-sized sub-vector in V ($h = 2^n/t$), generating a partial output equivalent in size to V. These partial outputs are then summed to obtain the output state vector W. Given

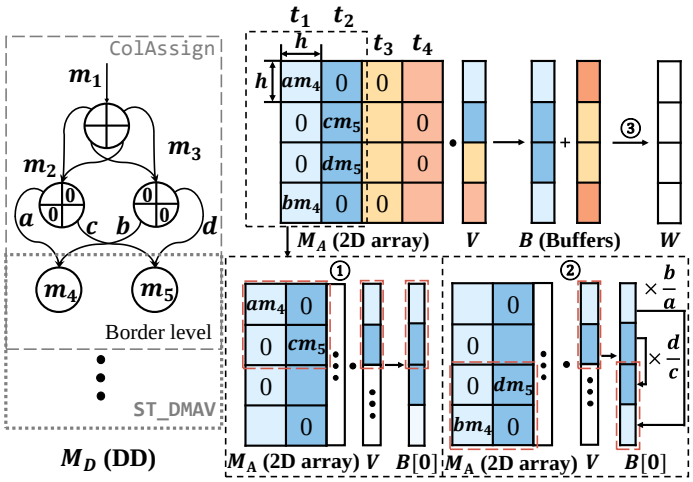


Fig. 9. Multi-threaded column-space DMAV on four threads, illustrated in the four colors. M_A is the 2D array representation of DD-based matrix M_D. The input state vector is V, and the resulting state vector is W. DD edges without labels have a weight of one by default.

the sparse nature of quantum gate matrices [37], the partial outputs from different threads often have non-overlapping non-zero sub-vectors. Thus, to save memory and time, multiple partial outputs can share a single memory *buffer*, and we sum the buffers to obtain the output state vector.

Like `RowAssign` in Algorithm 2, `ColAssign` divides the gate matrix into $h \times h$-sized sub-matrices, and vector V as well as the partial output vectors into h-sized sub-vectors. Each sub-matrix is paired with the corresponding sub-vectors, forming a multiplication task. To record the multiplication tasks for t threads, we use vectors v_M, v_P, and v_f to keep track of the sub-matrices' DD edges, start indices of sub-vectors in a partial output, and weight products assigned to each thread. We do not need to track the start indices of sub-vectors in V, as they are directly mapped to the thread index. Specifically, thread i computes $V[ih : (i+1)h]$. `ColAssign` has the following parameters (line 21): M_e is the DD edge corresponding to the current sub-matrix. v_M, v_P, and v_f are the 2D vectors tracking the multiplication tasks. f is the product of DD edge weights along the current DD traversal path, defaulting to $M_e.w$. u is the current thread index for task assignment, defaulting to 0. I_P is the start index of a sub-vector in a partial output, defaulting to 0. l denotes the DD level of the current node, initialized to the topmost level q_{n-1} (i.e., $l = n - 1$).

`ColAssign` returns if M_e is zero (line 22). Upon reaching the border level at $n - \log_2 t - 1$ (where `ColAssign` ends and `ST_DMAV` begins), we push the sub-matrix's edge M_e, the sub-vector's start index I_P, and the weight product f of thread u to the corresponding vectors (lines 23–25). At lines 26–28, we traverse the four outgoing edges of $M_e.n$ in column-major order and call `ColAssign` with the updated weight product, thread index, sub-vector's start index, and DD level, similar to `RowAssign` (line 16 in Algorithm 2).

In `MT_CS_DMAV`, after all threads have been assigned multiplication tasks (line 4), each thread is mapped to a buffer. Specifically, vector v_B associates each thread with a buffer index, which determines access to buffers in B, the vector of buffers (line 3). For each thread i, we check whether another thread j has a non-overlapping partial output (lines 5–6). If such a thread j exists, threads i and j will share a buffer (line 7); otherwise, thread i receives its own buffer (line 8). Then, each thread executes its multiplication tasks from `ColAssign`. For example, we focus on threads t_1 and t_2 in

Algorithm 3 Multi-threaded column-space DMAV (MT-CS-DMAV)

Require: M_e, edge representing a DD-based gate matrix; V and W, array-based input and output state vectors.

1: **function** MT_CS_DMAV(M_e, V, W)
2: $h \leftarrow 2^n/t$
3: $v_M = v_P = v_f = v_B \leftarrow \{\underbrace{\{\}, \cdots, \{\}}_{t}\}$; $B \leftarrow \{\}$
4: ColAssign(M_e, v_M, v_P, v_f)
5: **for** $i \in [0, t)$
6: **if not** Overlap($v_P[i]$, $v_P[j]$), $\exists j \in [0, i)$ **then**
7: $v_B[i] \leftarrow v_B[j]$
8: **else** $v_B[i] \leftarrow size(B)$; B.push($\{\underbrace{0, \cdots, 0}_{2^n}\}$)
9: **parallel for** $i \in [0, t)$
10: **for** $j \in [0, size(v_M[i]))$
11: $r \leftarrow cache[i]$.lookup($v_M[i][j].n$)
12: **if** $r \neq \{\}$ **then** $B[v_B[i]][v_P[i][j] : v_P[i][j] + h] \leftarrow$ SIMDMul($v_f[i][j]/r[0]$, $B[v_B[i]][r[1] : r[1] + h]$)
13: **else**
14: ST_DMAV($v_M[i][j]$,V, $B[v_B[i]]$, h, ih, $v_P[i][j]$, $v_f[i][j]$)
15: $cache[i]$.insert($v_M[i][j].n$, $\{v_f[i][j], v_P[i][j]\}$)
16: **parallel for** $i \in [0, t)$
17: **for** $j \in [0, size(B))$
18: $W[i \cdot h : (i+1) \cdot h] \leftarrow$ SIMDAdd($W[i \cdot h : (i+1) \cdot h]$, $B[j][i \cdot h : (i+1) \cdot h]$)
19: **end function**
20:
21: **function** ColAssign(M_e, v_M, v_P, v_f, $f = M_e.w$, $u = 0$, $I_P = 0$, $l = n - 1$)
22: **if** M_e **is** zero edge **then return**
23: **if** $l == n - \log_2 t - 1$ **then** /* Border level */
24: $v_M[u]$.push(M_e); $v_P[u]$.push(I_P);
25: $v_f[u]$.push(f); **return**
26: **else for** $j, i \in \{0, 1\}^2$
27: ColAssign($M_e.n.e[i][j]$, v_M, v_P, v_f, $f \cdot M_e.n.e[i]\ [j].w$, $u + \frac{j \cdot t}{2^{n-l}}$, $I_P + 2^l i$, $l - 1$)
28: **end function**

Figure 9 (in black dashed boxes). t_1 is assigned $a \cdot m_4 \cdot V[0:h]$ and $b \cdot m_4 \cdot V[0:h]$, while t_2 is assigned $c \cdot m_5 \cdot V[h:2h]$ and $d \cdot m_5 \cdot V[h:2h]$, where sub-vectors $V[0:h]$ and $V[h:2h]$ are the first two quarters in V. As these four tasks have non-overlapping outputs, they can share the same output buffer $B[0]$. At step ①, t_1 and t_2 execute $am_4 \cdot V[0:h]$ and $cm_5 \cdot V[h:2h]$ simultaneously. First, they search in caches (line 11). Finding no historical results, they directly call the ST_DMAV. Upon obtaining the result, t_1 caches m_4 with a pair consisting of m_4's weight product and the start index of the resulting sub-vector in t_1's partial output (i.e., $\{a, 0\}$). We only cache the left operand (sub-matrix DD node m_4) because the right operand (h-sized sub-vector in V) is the same for all assigned multiplication tasks on the same thread. Similarly, t_2 caches m_5 and $\{c, h\}$ (lines 14–15). At step ②, t_1 and t_2 execute $bm_4 \cdot V[0:h]$ and $dm_5 \cdot V[h:2h]$ simultaneously. Again, they check if m_4 and m_5 are cached, discovering historical results (line 11). t_1 finds an h-sized sub-vector in its partial output (stored in buffer $B[0]$) at start index 0 with weight a. It then multiplies this sub-vector by b/a using SIMD-enabled scalar multiplication and places it in the fourth quarter of the $B[0]$ (line 12). Likewise, t_2 performs a similar scalar multiplication. Finally, at step ③, we sum the buffers to form the final result W using multi-threading and SIMD (lines 16–19).

3.2.4 Optimized Multi-threaded Column-space DMAV. To mitigate the runtime and memory overhead of MT-CS-DMAV (Section 3.2.3), we eliminate its buffer requirement and propose an optimized multi-threaded column-space DMAV (MT-CS-DMAV*) in FlatDD-v2. To achieve this, each thread exclusively computes one unique multiplication task, while other multiplication tasks reuse its result through SIMD-enabled scaling. In this way, the historical results can be stored in a 2^n-sized vector (e.g., W) without requiring additional buffers.

We introduce function `Opt_MT_CS_DMAV` in Algorithm 4. `Opt_MT_CS_DMAV` performs DMAV in column space by multiplying a DD-based gate matrix (represented by the topmost edge M_e) by array-based vector V and storing the output state vector back into V. The array-based vector W is used to store historical results temporarily. At lines 2–4, we assign h-sized DMAV multiplication tasks to each thread using `ColAssign`, where $h = 2^n/t$. Then, we perform *initial DMAV*, where each thread performs the *first* assigned multiplication task and stores the h-sized sub-vector result in W (thread i writes to $W[ih : (i+1)h]$). This is illustrated in step ① in Figure 10: threads t_1, t_2, t_3, and t_4 first compute $am_4 \cdot V_0$, $cm_5 \cdot V_1$, $bm_4 \cdot V_2$, and $dm_5 \cdot V_3$, respectively, where $V_i = V[ih : (i+1)h]$. The results for these threads are stored in W_0, W_1, W_2, and W_3, where $W_i = W[ih : (i+1)h]$.

Next, we clear V to store the resulting output state vector (line 7), which is computed from W. Each thread derives the results of all multiplication tasks by scaling the result of its initial DMAV in W. We ensure that each thread computes only one unique multiplication task (i.e., the initial DMAV) with only one unique sub-matrix DD node and sub-vector, allowing tasks to reuse this computation. If this condition is not met, we do not invoke Algorithm 4 in the first place. We then divide V into h-sized sub-vectors, each of which accumulates scaled initial DMAV results (i.e., sub-vectors from W) to form a sub-vector in the output state. We use vectors v_{ini} and v_s to store the start index of the initial DMAV results and the scalars used in the scaling process for each corresponding sub-vector in V. For each multiplication task j that reuses i-th initial result (i.e., W_i), we store the start index ih and scalar $v_f[i][j]/v_f[i][0]$ to v_{ini} and v_s (lines 8–11). For example, in Figure 10 at step ②, W_0 is reused in tasks $am_4 \cdot V_0$ and $bm_4 \cdot V_0$, which are eventually added to sub-vectors V_0 and V_3, respectively, as indicated by vector v_P. Since $am_4 \cdot V_0$ is already computed and stored in W_0, it can be directly added to V_0 without scaling (i.e., with a scalar of 1). Thus, for sub-vector V_0, the start index is recorded as 0 in v_{ini}, and the scalar is recorded as 1 in v_s. For task $bm_4 \cdot V_0$, we scale W_0 by b/a and add the result to V_3. Thus, for sub-vector V_3, the start index is recorded as 0 in v_{ini}, and the scalar is recorded as b/a in v_s.

Finally, at lines 12–14, each thread i computes V_i via SIMD-enabled scalar multiplication and addition based on v_{ini} and v_s, as shown in step ③ of Figure 10, yielding the final result in V.

The advantage of MT-CS-DMAV* (Algorithm 4) over MT-CS-DMAV (Algorithm 3) is governed by the *density* of the column-space gate matrix. We define the density of a gate acting on an n-qubit register as

$$\rho = \frac{\mathrm{nnz}(G)}{2^n \times 2^n}, \tag{5}$$

where $\mathrm{nnz}(G)$ is the number of nonzero entries of the gate's matrix representation. When a gate is highly sparse, the partial products assigned to different threads touch disjoint regions of the output vector, so Algorithm 3 can share a small number of accumulation buffers b; its buffer cost $2^n b/(d \cdot t)$ and its memory footprint both stay small. As density increases, the partial products overlap, b grows, and the memory and final-reduction cost of Algorithm 3 grow proportionally. Algorithm 4 eliminates the density-dependent accumulation-buffer footprint entirely by writing partial results directly into disjoint output slices, so its peak memory is largely insensitive to ρ, although its runtime may still vary with fusion span, task count, reuse pattern, and DD structure. Section 4.6 quantifies this relationship: as ρ

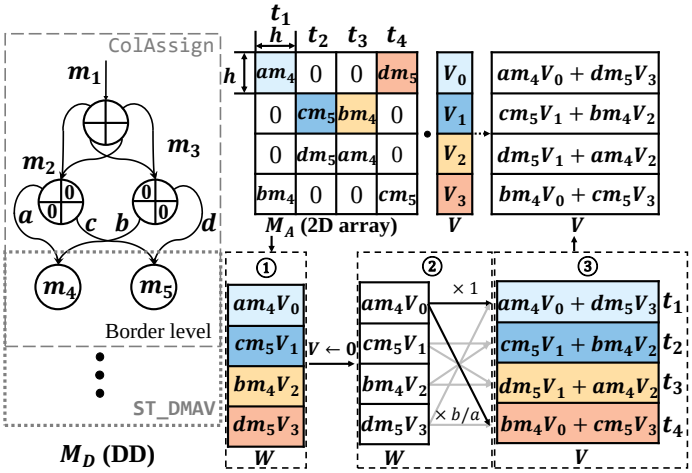


Fig. 10. Optimized multi-threaded column-space DMAV on four threads, illustrated in the four colors. M_A is the 2D array representation of DD-based matrix M_D. The input and output state vector is V, while W is an array-based vector that stores historical results. V_i and W_i represent $V[ih : (i+1)h]$ and $W[ih : (i+1)h]$, respectively. DD edges without labels have a weight of one by default.

increases by more than three orders of magnitude, the buffer count b and the peak memory of Algorithm 3 rise steeply, whereas Algorithm 4 stays flat (Table 5).

Algorithm 4 Optimized multi-threaded column-space DMAV (MT-CS-DMAV*)

Require: M_e, edge representing a DD-based gate matrix; V, array-based input and output state vector; W, array-based vector that stores historical results.

1: **function** Opt_MT_CS_DMAV(M_e, V, W)
2: $h \leftarrow 2^n/t$
3: $v_M = v_P = v_f \leftarrow \underbrace{\{\{\}, \cdots, \{\}\}}_{t}$
4: ColAssign(M_e, v_M, v_P, v_f)
5: **parallel for** $i \in [0, t)$ // Initial DMAV
6: ST_DMAV($v_M[i][0]$,V, W, h, ih, ih, $v_f[i][0]$)
7: $V \leftarrow \mathbf{0}$; $v_{ini} = v_s \leftarrow \underbrace{\{\{\}, \cdots, \{\}\}}_{t}$
8: **for** $i \in [0, t)$
9: **for** $j \in [0, size(v_M[i]))$
10: $v_{ini}[v_P[i][j]/h]$.push(ih)
11: $v_s[v_P[i][j]/h]$.push($\frac{v_f[i][j]}{v_f[i][0]}$)
12: **parallel for** $i \in [0, t)$
13: **for** $j \in [0, size(v_{ini}[i]))$
14: $V[i \cdot h : (i+1) \cdot h] \leftarrow$ SIMDAdd(SIMDMul($v_s[i][j]$, $W[v_{ini}[i][j] : v_{ini}[i][j] + h]$), $V[i \cdot h : (i+1) \cdot h]$)
15: **end function**

3.2.5 Computational Cost of DMAV Algorithms. Based on our observation, the MAC operation dominates DMAV computation. For example, within the ST_DMAV function, which accounts for 99.99% of Algorithm 2's runtime, the

only non-trivial computation is performed by line 6 in Algorithm 1, which executes a MAC operation. Therefore, we introduce the computational cost model based on the number of MAC operations.

First, it is necessary to determine the number of MAC operations in a ST-DMAV (Section 3.2.1). As shown in Figure 11, we use DFS to traverse DD nodes and a look-up table (MAC count table T) to track MAC operations for unique nodes, given that identical DD nodes incur the same number of MAC operations. The number of MAC operations of each node is the sum of that of its children, and the terminal node has one MAC operation. For each node, we traverse the outgoing edges from top to bottom and from left to right. Following this rule, at step ①, we go through m_1, m_2, m_3, and arrive at m_5. m_5 only has one outgoing edge to a terminal node with one MAC (i.e., $T(m_5) = 1$). At step ②, we proceed to m_3's lower-right outgoing edge, which also points to m_5. Thus, m_3 has two MAC operations (i.e., $T(m_3) = 2T(m_5) = 2$). Then, we return to m_2, and visit the upper-right edge pointing to m_4. Similarly, we derive that $T(m_6) = 1$ and $T(m_4) = 2$, at steps ③ and ④. The lower outgoing edges of m_2 connect to the previously visited nodes m_3 and m_4, giving us $T(m_2) = 2T(m_3) + 2T(m_4) = 8$ at step ⑤. Finally, at step ⑥, we return to m_1, obtaining $T(m_1) = 2T(m_2) = 16$. The total MAC count for this single-threaded DMAV is $T(m_1) = 16$.

Then, we model the computational cost for multi-threaded DMAV algorithms based on the number of MAC operations. We observe that, regardless of the algorithm used, the workload distribution among threads remains balanced. Therefore, the computational cost model evenly divides the number of MAC operations by t threads. We discuss the computational cost of multi-threaded DMAV Algorithms in Section 3.2.2, 3.2.3, and 3.2.4 as detailed below:

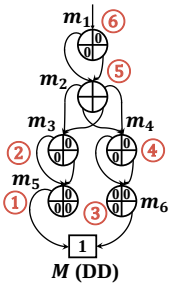
- **Multi-threaded row-space DMAV.** Suppose K_1 is the total number of MAC operations for `ST_DMAV` across all threads in Algorithm 2, Equation 6 models its computational cost C_{rs}.

$$C_{rs} = \frac{K_1}{t} \tag{6}$$

- **Multi-threaded column-space DMAV.** For Algorithm 3, we consider three costs. **(1) The cost of scalar multiplication.** We identify H cache hits, which are derived from repeated nodes connected by edges in v_M in Algorithm 3. Each hit corresponds to a scalar multiplication of size $h = 2^n/t$, leading to a total of $2^n H/t$ MAC operations for scalar multiplication. If SIMD computes d data elements simultaneously, on t threads, the cost of scalar multiplication is $2^n H/(d \cdot t^2)$. **(2) The cost of summing buffers.** Summing b buffers, each of size 2^n, incurs $2^n b$ MAC operations. When using SIMD on t threads, the cost of summing buffers is $2^n b/(d \cdot t)$. **(3) The cost unrelated to caching.** We count the MACs unrelated to caching in `ST_DMAV` across all threads, as in Figure 11, this time avoiding the addition of the repeated nodes mentioned in (1). Suppose there are K_2 MAC operations unrelated to caching, the corresponding cost is K_2/t. In summary, we model the computational cost C_{cs}, for Algorithm 3 as the following.

$$C_{cs} = \frac{K_2}{t} + \frac{2^n}{d \cdot t}\left(\frac{H}{t} + b\right) \tag{7}$$

- **Optimized multi-threaded column-space DMAV.** For Algorithm 4, we consider two costs. **(1) The cost of initial DMAV.** We count the MACs for the first `ST_DMAV` task (i.e., initial DMAV) across all threads, denoted as K_3, and the cost of initial DMAV is K_3/t. **(2) The cost of scalar multiplication.** We assume that there are a total of T DMAV multiplication tasks across all threads, with each thread handling an average of T/t tasks. For each task, we scale a sub-vector of size $h = 2^n/t$ from W to V in Algorithm 4. When using SIMD to compute d data elements simultaneously across t threads, the cost of scalar multiplication is $2^n T/(d \cdot t^2)$. In summary, we model the computational cost C^*_{cs} for Algorithm 4 as the following.



MAC count table T

	Node	#MACs
⑥	m_1	$2T(m_2) = 16$
⑤	m_2	$2T(m_3) + 2T(m_4) = 8$
②	m_3	$2T(m_5) = 2$
④	m_4	$2T(m_6) = 2$
①	m_5	1
③	m_6	1

Fig. 11. Counting the number of MAC operations of a DMAV.

(a) Sequential DMAV. (b) Gate fusion.

Fig. 12. Computation reduction from gate fusion.

$$C^*_{cs} = \frac{K_3}{t} + \frac{2^n \cdot T}{d \cdot t^2} \tag{8}$$

To minimize the total computational cost, we always select the multi-threaded DMAV algorithm with the lowest cost for each gate. In FlatDD-v1 [66], the computational cost for multi-threaded DMAV per gate is $\min\{C_{rs}, C_{cs}\}$. In FlatDD-v2, since optimized Algorithm 4 replaces Algorithm 3 for column-space DMAV, the computational cost per gate is $\min\{C_{rs}, C^*_{cs}\}$.

3.3 DMAV-Aware Gate-Fusion Algorithm

To further boost FlatDD's performance on large circuits, we propose a DMAV-aware gate-fusion algorithm that greedily fuses gates to reduce the total computational cost of DMAV.

Figure 12 shows the advantage of gate fusion. Consider two quantum gates M_1, M_2 in DD, there are two approaches to applying them consecutively to a state vector V using four threads. (1) **Sequential DMAV**: Multiply M_1 by V to obtain intermediate state W_1, and then multiply M_2 by W_1 to obtain the final state W_2, as shown in Figure 12a. (2) **Gate fusion**: Apply DD-based matrix-matrix multiplication (DDMM) to M_1 and M_2, fusing them into M_{21}, and then multiply M_{21} by V to obtain the final state W_2, as shown in Figure 12b. For simplicity, we evaluate two options to determine which one has a lower computational cost. In sequential DMAV, the two consecutive DMAVs result in a total computational cost of 128. In contrast, gate fusion performs one DDMM and one DMAV. The DMAV computational cost is 64. The DDMM calls 12 multiplications and 4 additions, which is negligible compared to the DMAV computational cost since each multiplication or addition only constructs one DD node. Therefore, gate fusion reduces computation compared to sequential DMAV.

(a) Sequential DMAV. (b) Gate fusion.

Fig. 13. Gate fusion can result in more computation.

However, gate fusion does not always reduce computation, as shown in Figure 13. In Figure 13a, sequential DMAV without gate fusion incurs a computational cost of 1536 using four threads. Conversely, in Figure 13b, gate fusion results in a DMAV computational cost of 2048 and 21 DDMM multiplication calls, also using four threads. In this case, sequential DMAV has a lower computational cost.

To reduce the total computational cost, we fuse two gates only when the fused gate has a smaller computational cost compared to sequential DMAV. Accordingly, we introduce our DMAV-aware gate fusion in Algorithm 5. Algorithm 5 operates on the group of remaining gates, G, after DD-to-array conversion. We initialize M_p (the previous gate matrix) to an identity DD matrix and set its DMAV computational cost (Section 3.2.5) C_p to 0 (line 2). Next, we iterate through G (line 3). For the i-th gate matrix $G[i]$, we calculate its DMAV computational cost, C_i. Then, we multiply $G[i]$ by M_p to form M_{ip}, and calculate its DMAV computational cost C_{ip} (line 4). If $C_i + C_p < C_{ip}$ fails to satisfy, we assign the fused gate M_{ip} and cost C_{ip} to the previous gate M_p and cost C_p (line 12). If $C_i + C_p < C_{ip}$, sequential DMAV is computationally cheaper than gate fusion (line 5). At lines 7–10, we introduce a look-ahead mechanism to determine whether further fusion of M_p can achieve an even lower computational cost. This mechanism helps prevent the greedy gate-fusion algorithm from being trapped at a locally sub-optimal point. We temporarily assign M_p to M_l as a copy (line 6). Then, we evaluate N gates ahead of i (i.e., $G[j]$, $j < \min\{i + N, size(G)\}$), sequentially fusing each $G[j]$ with M_l. If M_l has a lower cost than C_p, then we update M_p to M_l, set i to j, and update C_i to the cost of $G[j]$. After the look-ahead mechanism, we add M_p to the resulting gate group S. Before the next iteration, we assign the i-th gate $G[i]$ and its cost C_i to the previous gate M_p and its cost C_p (line 11). Finally, we return S at line 14.

3.4 Summary of Differences between FlatDD-v1 and FlatDD-v2

Table 1 summarizes the key differences between FlatDD-v1 and FlatDD-v2, listed as follows:

- **DD-to-array conversion:** FlatDD-v1 uses a multi-threaded implementation, whereas FlatDD-v2 adopts a task-parallel implementation. This design improves load balancing across threads and significantly enhances runtime performance.
- **DMAV:** Both versions employ multi-threaded row-space DMAV. However, for column-space DMAV, FlatDD-v2 significantly optimizes the algorithm used in FlatDD-v1 by mitigating the memory and runtime overhead introduced by intermediate buffers, thereby substantially improving the runtime performance of DMAV simulation.

Algorithm 5 DMAV-aware gate fusion

Require: G, group of remaining gates after DD-to-array conversion.
Ensure: S, group of gates after gate fusion.

function GateFuse(G)
  $M_p \leftarrow I_{DD}$; $C_p \leftarrow 0$; $i \leftarrow 0$
  while $i < size(G)$
    $C_i \leftarrow \text{cost}(G[i])$; $M_{ip} \leftarrow$DDMM$(G[i], M_p)$; $C_{ip} \leftarrow \text{cost}(M_{ip})$
    if $C_i + C_p < C_{ip}$ **then**
      $M_l \leftarrow M_p$; $C_l \leftarrow C_p$; $j \leftarrow i$
      while $j < \min\{i + N, size(G)\}$ // look ahead
        if $C_l \leq C_p$ **then**
          $M_p \leftarrow M_l$; $C_i \leftarrow \text{cost}(G[j])$; $i \leftarrow j$
        $M_l \leftarrow$DDMM$(G[j], M_l)$; $C_l \leftarrow \text{cost}(M_l)$; $j \leftarrow j + 1$
      S.push(M_p); $C_p \leftarrow C_i$; $M_p \leftarrow G[i]$
    else $M_p \leftarrow M_{ip}$; $C_p \leftarrow C_{ip}$
    $i \leftarrow i + 1$
  return S
end function

Table 1. Summary of differences between FlatDD-v1 and FlatDD-v2, highlighting key design improvements.

Component	FlatDD-v1	FlatDD-v2
DD-to-array conversion	Multi-threaded	Task-parallel
DMAV	Multi-threaded row-space DMAV + multi-threaded column-space DMAV	Multi-threaded row-space DMAV + optimized multi-threaded column-space DMAV
Computational cost modeling	Multi-threaded row-space DMAV + multi-threaded column-space DMAV	Multi-threaded row-space DMAV + optimized multi-threaded column-space DMAV
Gate fusion	Greedy	Greedy + look-ahead mechanism

- **Computational cost modeling:** In both versions, we model the computational cost of multi-threaded row-space DMAV. However, the two versions use different computational cost models for multi-threaded column-space DMAV, corresponding to their different algorithmic designs.
- **Gate fusion:** FlatDD-v2 incorporates a new look-ahead mechanism into the DMAV-aware gate-fusion algorithm, which helps prevent greedy gate fusion from being trapped in a locally sub-optimal solution.

4 Experimental Results

In this section, we evaluate the performance of FlatDD-v2 using 12 commonly used quantum circuits from `QASMBench` [76], `MQT Bench` [94] and `Quantum supremacy` [8]. These benchmarks span both regular and irregular structures. First of all, we compare the runtime and memory performance of FlatDD-v2 and FlatDD-v1 against two state-of-the-art QCSs (Section 4.2). Then, we demonstrate the scalability of FlatDD-v2 over increasing numbers of threads and qubits (Section 4.3). We then evaluate the effectiveness of our parallel DD-to-array conversion algorithm (Section 4.4) and DMAV algorithms (Section 4.5), and quantify how sparsity affects DMAV performance (Section 4.6). In Section 4.7, we evaluate our DMAV-aware gate-fusion algorithm on deep circuits with thousands of gates. Lastly, to better understand the advantages and limitations of FlatDD relative to tensor network-based methods, we compare FlatDD-v2 with Qiskit Aer's MPS simulator in Section 4.8. All the experiments were primarily conducted on a Ubuntu 22.04.5 LTS machine

with 16 Intel i7-11700 logical CPU cores at 2.50 GHz, and 128 GB memory capacity.[2] Due to a change in the first author's affiliation, however, the experiments reported in Table 7, Table 8, and Figure 16 were conducted on a different platform. Specifically, those data were collected on an Ubuntu 24.04.4 LTS machine equipped with 48 AMD EPYC 7R32 logical CPU cores at 2.8 GHz and 192 GB of memory. In addition, the newly added controlled ablation in Table 3 and the sparsity study in Table 5 were collected on a third machine, an Ubuntu 26.04 LTS system with 16 Intel Xeon 6982P-C logical CPU cores (8 cores per socket, 2 threads per core) at 3.9 GHz and 64 GB of memory. Because these two tables come from a different platform, their absolute runtimes are not directly comparable to those in Table 2 and Table 4; within each individual table, however, all configurations were run on the same machine, so the reported relative comparisons remain valid. We compile all programs with optimization flag `-O3` enabled. All SIMD operations are executed using Intel's `AVX2` [64] vector instructions. For data with exponential difference, we measure the average in geometric mean. We implement our task graph using `Taskflow` [55].

4.1 Baselines

Given the large number of QCSs, it is not possible to compare FlatDD-v2 with all of them. Instead, in addition to FlatDD-v1, we consider two representative QCSs, DDSIM [105] and Quantum++ [22], for two reasons: (1) Both DDSIM and Quantum++ are highly optimized and have demonstrated superior performance over existing QCSs. DDSIM introduces a DD-based compact representation of gate matrices and state vectors, as well as an efficient data structure for handling complex numbers [104]. On the other hand, Quantum++ leverages `OpenMP` [3] to enable multi-threaded simulation based on `Eigen` [26] array and matrix data structures. (2) Both DDSIM and Quantum++ are open-source [6, 93] and widely used by the community, allowing us to fairly study and reason their results.

4.2 Overall Performance Comparison

Table 2 compares the overall performance of FlatDD-v2 with FlatDD-v1, DDSIM and Quantum++. In this experiment, we do not incorporate the proposed gate-fusion algorithm but focus on the full-state simulation workload itself. We run FlatDD-v2, FlatDD-v1 and Quantum++ using 16 threads and run DDSIM using one thread as DDSIM does not support multi-threading. For all FlatDD-v2 and FlatDD-v1 runs, we set $\beta = 0.9$ and $\epsilon = 2$, as these values are determined to be effective across multiple quantum circuits. In this experiment, different circuits do not require circuit-specific tuning; the same β and ϵ parameters are used for all reported results. Unless otherwise specified, all runtime and memory are measured in seconds (s) and megabytes (MB). We terminate the runs that take longer than 24 hours. We measure memory usage by recording the maximum resident set size (RSS) of our program using `/bin/time`.

We report only the memory usage of FlatDD-v2 in Table 2, as it is similar to that of FlatDD-v1 in this experiment. Note that the similar memory usage of FlatDD-v1 and FlatDD-v2 in this experiment does not contradict the discussion in Section 3.2.4. FlatDD-v1 selects between MT-RS-DMAV and MT-CS-DMAV at runtime. However, when gate fusion is not applied, MT-CS-DMAV is generally not selected because its buffer-related overhead outweighs its limited computation-reuse benefit. Therefore, FlatDD-v1 effectively uses MT-RS-DMAV in this setting, which does not allocate additional buffers. On the other hand, FlatDD-v2 selects between MT-RS-DMAV and the optimized MT-CS-DMAV* at runtime, and the optimized version also avoids additional buffers. As a result, both versions exhibit similar memory usage in this experiment. The memory-usage advantage of FlatDD-v2 becomes more apparent when gate fusion is enabled, as later shown in Section 4.5.

[2]Since we are using a different machine, the baseline results differ from those reported in the ICPP version [66], which uses 64 logical CPU cores on a dual-socket, two-NUMA-node system with two Intel Xeon Gold 6226R CPUs.

On average, FlatDD-v2 is 2.8%, 54.35× and 19.44× faster over FlatDD-v1, DDSIM and Quantum++ on all circuits. In terms of memory usage, FlatDD-v2 is 1.78× and 1.91× less than DDSIM and Quantum++. DDSIM demonstrates exceptional performance on regular circuits such as Adder and GHZ state. For instance, the circuit Adder has a very regular distribution of state amplitudes throughout the simulation; DDSIM takes less than 1s while Quantum++ takes 1195.84s. However, when the regularity does not appear frequently, such as DNN, VQE, and Quantum supremacy, DDSIM becomes significantly slower than FlatDD-v2, FlatDD-v1 and Quantum++. For example, DDSIM is 528.72×, 478.67× and 96.22× slower than FlatDD-v2, FlatDD-v1 and Quantum++ when simulating the 20-qubit DNN.

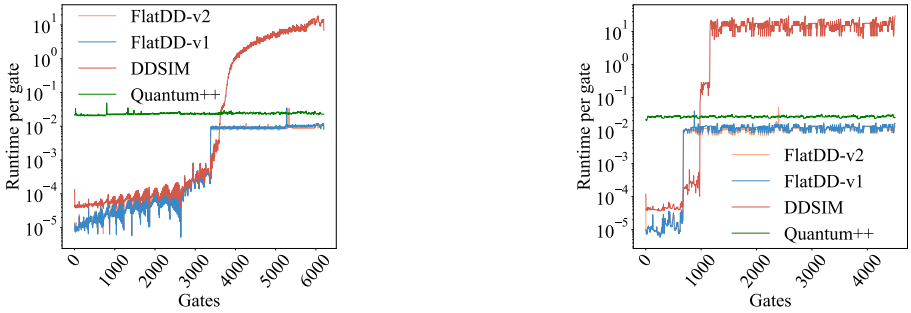
Although DDSIM and Quantum++ have their own strength in simulating certain circuits, FlatDD-v2 demonstrates a consistent performance advantage on all. For highly regular circuits, such as Adder, and GHZ state, where DDSIM performs extremely well, FlatDD-v2 also achieves fast runtime (< 1 s). This is because FlatDD-v2 does not switch from DDSIM to DMAV during the simulation. The only reason that FlatDD-v2 can be slower than DDSIM, such as simulating the 23-qubit GHZ state, is the overhead of DD size calculation and conversion timing checking (Section 3.1.1). For irregular circuits, such as 20-qubit DNN and quantum supremacy circuit, FlatDD-v2 can still take advantage of DD-based state vector to a certain extent before the state vector turns irregular. After this turning point, as observed from Figure 14a and Figure 14b, the runtime of DDSIM will significantly increase whereas FlatDD-v2 cleverly converts to multi-threaded DMAV and stays stable. Additionally, after the turning point, FlatDD-v2 is faster than Quantum++ primarily due to the efficient indexing pattern of our DMAV (see Section 3.2.1).

Compared to FlatDD-v1, FlatDD-v2 is up to 10.46% faster on the 20-qubit DNN and 2.8% faster on average across all circuits. This is because FlatDD-v2 reuses computation in DD-based gates through caching, whereas FlatDD-v1 does not, due to the higher caching overhead of its MT-CS-DMAV. As a result, FlatDD-v1 applies MT-RS-DMAV without caching across all gates, making it slower than FlatDD-v2. In contrast, FlatDD-v2, utilizes both MT-RS-DMAV and MT-CS-DMAV*, which supports caching, throughout the circuit. As demonstrated Figure 14, FlatDD-v2 and FlatDD-v1 follow the same trend, except that for certain gates, FlatDD-v2 achieves lower runtime.

The memory usage of FlatDD-v2 is comparable to DDSIM and Quantum++. When the circuit is small, Quantum++ consumes the smallest memory compared with FlatDD-v2 and DDSIM, both of which require additional data storage for DDs. For example, when simulating the 16-qubit DNN, Quantum++ needs only 9.0 MB, whereas FlatDD-v2 and DDSIM need 34.7 MB and 73.7 MB, respectively. On the other hand, when the circuit is large, Quantum++ consumes significantly higher memory than FlatDD-v2 and DDSIM. For instance, when simulating the 25-qubit KNN, Quantum++ needs 1577.37 MB, while FlatDD-v2 and DDSIM only need 1079.47 MB and 439.92 MB, respectively. If the state vector is highly regular (e.g., Adder, GHZ state), FlatDD-v2 and DDSIM have similar memory usage, as FlatDD-v2 does not switch from DDSIM to DMAV. However, if the state vector contains high irregularity (e.g., DNN, VQE, Quantum supremacy), FlatDD-v2 has smaller memory usage than DDSIM because it will switch from DDSIM to DMAV to reduce DD overheads. Lastly, if the state vector is irregular in large circuits, FlatDD-v2 consumes smaller memory than both DDSIM and Quantum++. For example, when simulating the 26-qubit quantum supremacy circuit, FlatDD-v2 consumes 2135.46 MB memory, while DDSIM and Quantum++ consume 16801.2 and 3157.49 MB memory. When the quantum state becomes highly irregular, FlatDD-v2 transitions from a purely DD-based simulation to DMAV-based execution, in which the array component dominates the overall memory usage. DMAV uses two arrays for input and output. For example, for the 31-qubit KNN, the theoretical memory requirement of DMAV is $2^{31} \times 2 \times (8 + 8) = 68.71$ GB, where the first factor of 2 comes from input and output, and the second factor $(8 + 8)$ accounts for the double-precision real and imaginary parts of each amplitude. Hence, the reported FlatDD-v2 memory usage of 67.14 GB is consistent with this theoretical value.

Table 2. Comparison of simulation runtime and memory among FlatDD-v2, FlatDD-v1, DDSIM, and Quantum++ on 12 widely used circuits that exhibit different regularity structures.

Circuits			FlatDD-v2		FlatDD-v1 [66]		Quantum++ [22]			DDSIM [105]		
Name	n	Gates	Runtime	Memory	Runtime	Runtime improvement (%)	Runtime	Speed-up	Memory	Runtime	Speed-up	Memory
DNN	16	2032	**1.176**	34.7	1.179	0.25%	2.45	2.08×	**9.0**	37.02	31.48	73.7
	20	6214	**26.86**	76.9	29.67	10.46%	147.6	5.49×	**60.26**	14202.2	528.72	523.24
	25	9644	**2057.42**	**1091.44**	2200.58	6.96%	10475	5.09×	1586.9	> 24 h	> 41.99×	≥ 4886.05
Adder	28	117	0.011	30.5	0.011	≈ 0%	1195.84	105386×	12587.8	**0.004**	0.37×	**29.9**
GHZ state	23	46	0.008	30.6	0.008	≈ 0%	6.62	785.06×	398.2	**0.001**	0.17×	**30.0**
VQE	16	95	**0.0509**	32.54	0.0512	0.51%	0.098	1.93×	**8.2**	50.60	993.35×	54.18
KNN	25	39	**3.69**	1079.47	3.72	0.95%	84.82	22.99×	1577.37	358.58	97.18×	**439.92**
	31	48	**403.29**	67140.01	403.45	≈ 0%	8318.88	20.63×	100667.7	> 24 h	> 214.24×	≥ 2758.79
Swap test	25	39	**3.70**	1079.6	3.71	0.47%	85.37	23.1×	1577.7	1156.04	312.85×	**739.86**
Quantum supremacy	20	4500	**44.49**	67.48	48.26	8.47%	116.11	2.61×	**59.136**	53128.8	1194.18×	719.41
	24	5560	**926.37**	**560.83**	977.64	5.53%	3006.81	3.24×	797.18	> 24 h	> 93.27×	≥ 5795.84
	26	5990	**4091.16**	**2135.46**	4210.84	2.92%	13797	3.37×	3157.49	> 24 h	> 21.12×	≥ 16801.2
Average			**9.029**	**301.33**	9.282	2.8%	175.56	19.44×	575.863	> 490.8	> 54.35×	≥ 537.118



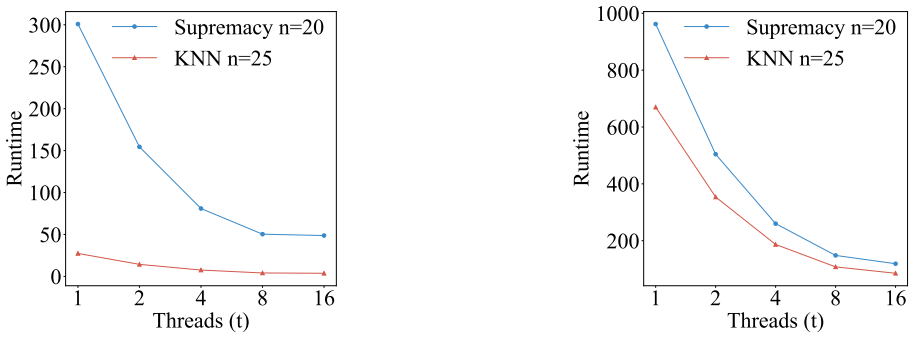
(a) DNN $n = 20$. (b) Supremacy $n = 20$.

Fig. 14. Runtime comparison among FlatDD-v2, FlatDD-v1, DDSIM, and Quantum++ per gate.

Note that in the original ICPP version [66], the experiments were conducted on a dual-socket, two-NUMA-node system with two Intel Xeon Gold 6226R CPUs. Although this platform provides stronger aggregate multicore resources, the 16 execution threads and their memory allocations were not explicitly pinned to the same NUMA node. Consequently, the computation could incur remote-memory accesses across NUMA nodes, which introduce additional latency and may reduce performance. By contrast, the current platform is a single-socket system, which avoids cross-NUMA memory accesses. In addition, although the current processor has only 8 physical cores, its 16 logical threads can still provide competitive performance because hyper-threading helps hide memory-access stalls. Therefore, the stronger nominal hardware specification of the ICPP platform does not necessarily translate into better runtime for this workload.

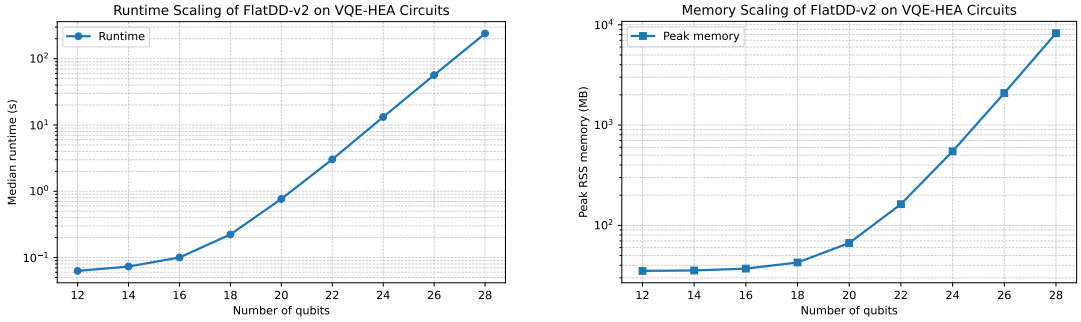
4.3 Scalability of FlatDD-v2

4.3.1 Scalability under Increasing Number of Threads. Figure 15 illustrates the runtime of FlatDD-v2 and Quantum++ at different numbers of threads on two circuits (Quantum supremacy and KNN). When the number of threads increases, FlatDD-v2 can complete the simulation faster, as illustrated in Figure 15a. For instance, at eight threads, FlatDD-v2 is 6.69× faster than one thread on KNN. FlatDD-v2's performance saturates at about 16 threads. Likewise, in Figure 15b, Quantum++ shows a similar trend.



(a) FlatDD-v2. (b) Quantum++.

Fig. 15. Runtime scalability of FlatDD-v2 and Quantum++ under different numbers of threads.

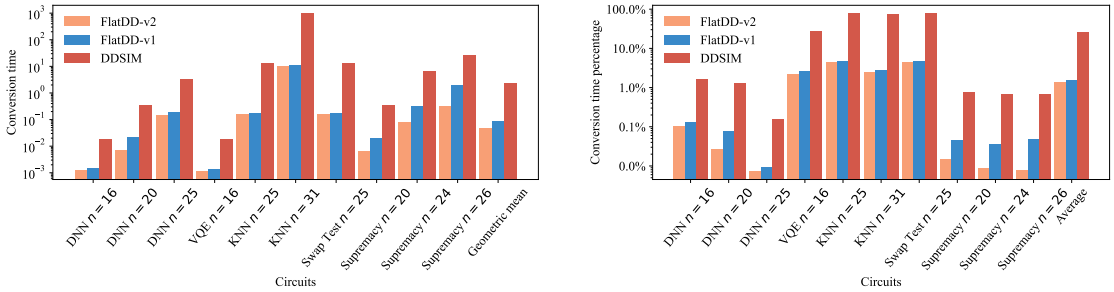


(a) (b)

Fig. 16. Scalability of FlatDD-v2 under circuits with increasing number of qubits. (a) Runtime scaling of FlatDD-v2. (b) Memory scaling of FlatDD-v2.

Generally, the main bottleneck is memory bandwidth rather than computation. After FlatDD converts the state representation from DD to DMAV, the dominant operation becomes matrix-vector multiplication over a flat state-vector array. Although the DD-based gate matrix still helps reduce indexing overhead and exploit gate sparsity, the array-based state vector must still be read and written repeatedly. Therefore, the computation-to-memory-access ratio of DMAV is relatively low, making FlatDD memory bandwidth-bound.

4.3.2 Scalability under Increasing Number of Qubits. Figure 16 shows the runtime and the peak memory usage of FlatDD-v2 on VQE-HEA circuits with the number of qubits ranging from 12 to 28. As shown in Figure 16, FlatDD-v2's runtime increases from 63 ms at 12 qubits to 239.79 s at 28 qubits, while its peak memory usage increases from 36.02 MB to 8424.82 MB. This trend is consistent with the fact that, once FlatDD switches to DMAV, it still follows state-vector-style memory scaling.



(a) Conversion time comparison.

(b) Conversion percentage in total runtime comparison.

Fig. 17. Comparison among FlatDD-v2's task-parallel DD-to-array conversion, FlatDD-v1's multi-threaded DD-to-array conversion, and DDSIM's sequential DD-to-array conversion.

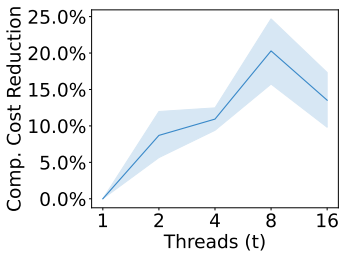
4.4 Evaluation of DD-to-array Conversion Algorithm

Figure 17 compares the efficiency of FlatDD-v2's task-parallel DD-to-array conversion algorithm with FlatDD-v1's multi-threaded conversion and DDSIM's sequential conversion on 10 quantum circuits. In this experiment, we integrate DDSIM's and FlatDD-v1's conversion algorithms into FlatDD-v2. Figure 17a compares the conversion time, and Figure 17b compares the cost of conversion in terms of its percentage in the total simulation time. We set threshold level $L_C = n - \log_2 t - 2$, as it provides good performance across all circuits on our machine. Here, $n - \log_2 t - 1$ is the border level (see Section 3.2.2) that assigns tasks for the t threads. We choose $L_C = n - \log_2 t - 2$, i.e., one level finer, so that the number of generated tasks exceeds the number of threads. This improves load balancing when the DD structure is irregular. The conversion algorithms of FlatDD-v1 and FlatDD-v2 run on 16 threads.

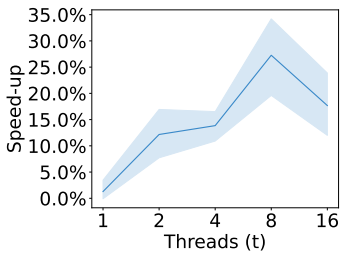
As shown in Figure 17a, FlatDD-v2 outperforms FlatDD-v1 and DDSIM in converting DDs to arrays on all circuits. For instance, FlatDD-v2's DD-to-array algorithm is 80.56× faster than DDSIM on KNN of 25 qubits; on average, FlatDD-v2 is 47.81× faster, which is attributed to task parallelism and SIMD parallelism. Compared with FlatDD-v1, FlatDD-v2 achieves up to 6.17× speedup (1.88× on average) due to its balanced workload distribution among threads. In terms of conversion cost, as shown in Figure 17b, FlatDD-v2's DD-to-array algorithm takes no more than 4.37% of the total runtime, while DDSIM can take up to 78.9% (Swap test of 25 qubits). This result highlights the efficiency of FlatDD-v2's task-parallel DD-to-array conversion algorithm.

4.5 Evaluation of DMAV Algorithms

Figure 18 demonstrates the benefit of MT-CS-DMAV in terms of reduced computational cost and speed-up across different numbers of threads. We run FlatDD-v1, which includes both MT-CS-DMAV and MT-RS-DMAV, against an implementation of FlatDD-v1 with only MT-RS-DMAV on the six largest quantum circuits: DNN ($n = 16, 20, 25$) and quantum supremacy ($n = 20, 24, 26$). We plot the results in a region (marked in blue), with a line showing the average. As shown in Figure 18a and Figure 18b, MT-CS-DMAV can reduce the computational cost as the number of threads increases. For instance, with 16 threads, we can achieve 13.53% reduction. A similar trend can be observed in the speed-up plot (Figure 18b). For example, MT-CS-DMAV contributes to an average of 17.67% speed-up at 16 threads where the performance saturates (see Figure 15). These results highlight the benefit of MT-CS-DMAV across different thread counts due to caching and reusing historical results.



(a) Computational cost.



(b) Speed-up.

Fig. 18. Performance benefit from MT-CS-DMAV over different numbers of threads on various quantum circuits.

Table 3. Controlled ablation isolating MT-CS-DMAV* (Algorithm 4) over MT-CS-DMAV (Algorithm 3). All settings are fixed except the column-space DMAV algorithm, with *gate fusion enabled*. Runtime is in seconds and memory in MB.

Circuits		DNN		Quantum supremacy	
Qubits (n)		20	25	24	26
MT-CS-DMAV	Time	7.87	248.70	217.35	408.42
	Memory	215.87	5425.24	2736.22	10800.88
MT-CS-DMAV*	Time	6.77	195.52	175.71	316.75
	Runtime improvement (%)	13.9%	21.4%	19.2%	22.4%
	Memory	80.26	1073.65	560.45	2096.21
	Memory reduction	2.69×	5.05×	4.88×	5.15×

Controlled ablation of MT-CS-DMAV*. We next isolate the contribution of the optimized column-space DMAV (MT-CS-DMAV*, Algorithm 4) relative to the original column-space DMAV (MT-CS-DMAV, Algorithm 3). To enable a fair single-variable comparison, all other settings are held fixed, namely the same circuits, the same 16 threads, the same DD-to-array conversion, and *gate fusion enabled*, so that the column-space DMAV algorithm is the only difference between the two configurations. Table 3 reports the results on the four largest quantum circuits: DNN ($n = 20, 25$) and Quantum Supremacy ($n = 24, 26$). Under this controlled setting, MT-CS-DMAV* reduces peak memory by 2.69–5.15× (4.44× on average) and improves runtime by up to 22.4% (1.24× on average), indicating that the buffer-elimination design of Algorithm 4 is the direct source of the DMAV-level gain.

Cumulative system-level comparison. We then compare MT-CS-DMAV* and MT-CS-DMAV at the system level, i.e., within the complete FlatDD-v2 and FlatDD-v1 pipelines. Table 4 evaluates FlatDD-v2, which uses MT-CS-DMAV*, against FlatDD-v1, which uses MT-CS-DMAV, on the four largest quantum circuits: DNN ($n = 20, 25$) and Quantum Supremacy ($n = 24, 26$), while enabling gate fusion. Unlike the controlled ablation in Table 3, this comparison reflects the *cumulative* effect of all FlatDD-v2 improvements (task-parallel conversion, the look-ahead gate-fusion mechanism, and MT-CS-DMAV*) rather than the isolated contribution of Algorithm 4. The results in Table 4 are obtained with gate fusion enabled. Gate fusion increases reusable computation within each DD-based gate, ensuring that FlatDD-v1 utilizes buffer-based MT-CS-DMAV to reuse historical results. Because MT-CS-DMAV requires additional buffers, its memory usage is substantially higher than that observed in Section 4.2. Overall, FlatDD-v2 achieves up to 40% runtime improvement and reduces memory by up to 4.69×. The accompanying reduction in computational cost (up to 36%) is contributed by FlatDD-v2's look-ahead gate fusion, which produces cheaper fused gates; as shown in the controlled

Table 4. Comparison of simulation runtime (s), computational cost, and memory usage (MB) between FlatDD-v2 and FlatDD-v1.

Circuits		DNN		Quantum supremacy	
Qubits (n)		20	25	24	26
FlatDD-v1	Time	3.27	151.14	107.11	478.34
	Cost	2.8×10^7	1.2×10^9	8.9×10^8	3.9×10^9
	Memory	205.7	5292.92	2670.32	10534.47
FlatDD-v2	Time	3.07	127.41	76.95	341.48
	Runtime improvement (%)	6%	19%	39%	40%
	Cost	2.4×10^7	1.1×10^9	6.6×10^8	2.9×10^9
	Cost reduction (%)	16%	14%	36%	36%
	Memory	96.27	1141.77	675.02	2247.73
	Memory reduction	2.14×	4.64×	3.96×	4.69×

ablation (Table 3), MT-CS-DMAV* and MT-CS-DMAV perform the same mathematical DMAV operation and produce bit-identical state vectors, so this cost reduction stems from the fused-gate workload rather than from the choice of column-space kernel. The runtime and memory savings of MT-CS-DMAV* over MT-CS-DMAV are therefore primarily attributable to its different buffering and reduction strategy rather than to a different fused-gate computational cost. Together, Table 3 and Table 4 demonstrate both the isolated efficiency of MT-CS-DMAV* and its benefit within the full FlatDD-v2 system.

4.6 Impact of Sparsity on DMAV

To isolate the sparsity effect predicted in Section 3.2.4, we sweep the density of the executed column-space gates while holding the circuit, qubit count, thread count, and cost model fixed. We control density through the gate-fusion span s: fusing s consecutive gates into a single decision diagram produces a denser fused gate matrix, so increasing s walks the workload from many sparse gates toward fewer, denser ones. For every executed column-space gate we record its density ρ and, for MT-CS-DMAV (Algorithm 3), the number of accumulation buffers b it allocates. Both column-space kernels are run from the same binary, differing only in their buffering strategy, and both produce bit-identical state vectors.

Table 5 reports the results on `supremacy_n20`. As the fusion span grows from 1 to 16, the mean density of the column-space gates rises by more than three orders of magnitude (2.49×10^{-6} to 6.67×10^{-3}), and the maximum buffer count required by MT-CS-DMAV saturates at 16. Over the same range, the peak memory of MT-CS-DMAV increases from roughly 133 MB to 336 MB, whereas the peak memory of MT-CS-DMAV* (Algorithm 4) stays essentially constant at 69–80 MB. This diverging pair of curves directly confirms the mechanism of Section 3.2.4: the buffer footprint of MT-CS-DMAV grows with density, while the buffer-free design of MT-CS-DMAV* makes its memory independent of ρ. On `dnn_n20` (Table 5, lower block), fusion reduces the number of column-space gates without raising their per-gate density, so it acts as a fixed-density control; even there, the memory of MT-CS-DMAV exceeds that of MT-CS-DMAV* at every point, showing that the buffer overhead is always present.

We stress that the sparsity effect is a property of the column-space kernel rather than of gate fusion: fusion is used here only as a convenient, workload-faithful knob to vary ρ. The buffer-free design of MT-CS-DMAV* is what removes the density-dependent accumulation-buffer footprint, rendering its peak memory largely insensitive to ρ; its runtime may still vary with fusion span, task count, reuse pattern, and DD structure.

Table 5. Effect of gate-matrix density, swept via the fusion span s, on the two column-space kernels ($t = 16$). Density ρ is the mean over executed column-space gates; $b_{\max}$ is the maximum buffer count allocated by MT-CS-DMAV (Algorithm 3), while MT-CS-DMAV* (Algorithm 4) allocates none. Peak memory (MB) is measured with `/usr/bin/time -v`, and runtime (rt) is in seconds. On `dnn_n20` the mean density stays essentially flat ($\approx 2 \times 10^{-6}$) because fusion reduces the *number* of column-space gates rather than their per-gate density, so it serves as a fixed-density control.

`supremacy_n20` (20 qubits)							
s	col-gates	mean ρ	$b_{\max}$	Alg. 3 mem	Alg. 4 mem	Alg. 3 rt	Alg. 4 rt
1	225	2.49×10^{-6}	4	133	69	43.3	22.8
2	151	3.45×10^{-6}	8	200	73	23.1	25.3
3	105	5.12×10^{-6}	8	197	69	21.7	19.6
5	76	1.14×10^{-5}	16	328	73	21.2	19.5
8	51	4.57×10^{-5}	16	328	73	41.9	40.0
12	45	9.55×10^{-4}	16	329	73	226.0	185.1
16	29	6.67×10^{-3}	16	336	80	424.4	598.3
`dnn_n20` (20 qubits, fixed-density control)							
1	592	1.91×10^{-6}	2	102	70	27.3	20.9
2	513	1.91×10^{-6}	2	102	70	21.1	15.8
3	405	1.94×10^{-6}	4	134	70	16.9	12.9
5	184	2.37×10^{-6}	4	134	70	10.5	8.3
8	92	2.72×10^{-6}	4	137	73	7.4	6.1
12	80	2.50×10^{-6}	4	137	73	6.0	5.1
16	35	3.11×10^{-6}	4	137	73	5.1	4.7

Table 6. Comparison of simulation runtime (s) and computational cost among FlatDD-v2 with DMAV-aware gate fusion (ours), FlatDD-v2 without gate fusion, and FlatDD-v2 with `k-operations` [106] on six deep circuits (> 1000 gates). Red. is the reduction in computational cost.

Circuits			FlatDD-v2 with DMAV-aware gate fusion (ours)		FlatDD-v2 without gate fusion				FlatDD-v2 with `k-operations` [106]			
Name	n	Gates	Runtime	Cost	Runtime	Speed-up	Cost	Red.	Runtime	Speed-up	Cost	Red.
DNN	16	2032	**0.28**	$\mathbf{4.4 \times 10^5}$	1.18	4.17×	7.8×10^6	17.67×	0.38	1.34×	3.1×10^6	7.06×
	20	6214	**3.07**	$\mathbf{2.4 \times 10^7}$	26.86	8.75×	2.2×10^8	9.45×	10.53	3.4×	9.7×10^7	4.1×
	25	9644	**127.41**	$\mathbf{1.1 \times 10^9}$	2057.42	16.15×	1.6×10^{10}	14.57×	659.97	5.18×	5.0×10^9	4.60×
Quantum supremacy	20	4500	**3.92**	$\mathbf{3.4 \times 10^7}$	44.49	11.35×	3.7×10^8	10.91×	28.24	7.21×	2.5×10^8	7.51×
	24	5560	**76.95**	$\mathbf{6.6 \times 10^8}$	926.37	12.04×	7.2×10^9	11.0×	597.88	7.77×	5.1×10^9	7.77×
	26	5990	**341.48**	$\mathbf{2.9 \times 10^9}$	4091.16	11.98×	3.1×10^{10}	10.90×	2749.49	8.05×	2.3×10^{10}	8.12×
Average			**14.99**	$\mathbf{9.5 \times 10^7}$	149.04	9.94×	1.2×10^9	12.13×	70.43	4.70×	6.0×10^8	6.31×

4.7 Evaluation of DMAV-aware Gate Fusion

In this experiment, we evaluate the performance advantage of our gate-fusion algorithm by running FlatDD-v2 on the six deepest circuits (DNN and supremacy) with and without gate fusion. We compare the result with `k-operations` [106], a particularly effective gate-fusion algorithm designed for DD-based simulators. As shown in Table 6, our gate-fusion algorithm achieves 9.94× and 4.70× speed-up in terms of geometric mean when compared with FlatDD-v2 without gate fusion and `k-operations`, respectively. Our gate-fusion algorithm also reduces computational cost by 12.13× and 6.31× when compared with FlatDD-v2 without gate fusion and `k-operations`, respectively. This is because our gate-fusion algorithm always fuses quantum gates that reduce computational cost, and our look-ahead mechanism prevents it

Table 7. Runtime overhead of gate fusion in FlatDD for three representative benchmark circuits. The table reports the absolute gate-fusion runtime and its percentage of the total simulation time.

Circuits	KNN	VQE-HEA	Swap test
Number of qubits (n)	20	20	22
Gate fusion runtime (ms)	3.07	32.4	50.4
Gate fusion runtime percentage (%)	1.19	4.28	5.05

Table 8. Comparison of FlatDD-v2 and Qiskit Aer's MPS simulator on eight representative circuits. The faster runtime in each row is shown in bold.

Circuit	n	Parameter	MPS max bond dimension	FlatDD-v2 runtime (s)	MPS runtime (s)
DNN	16	depth = 12	256	**0.216**	24.820
DNN	20	depth = 8	1024	**1.113**	437.865
Supremacy	18	depth = 8	512	**0.254**	12.244
Swap test	18	layer = 3	24	**0.188**	25.840
VQE-HEA	20	layer = 32	8	0.758	**0.006**
QAOA	20	layer = 2	11	0.861	**0.520**
GHZ	22	depth = 1	2	0.072	**0.002**
KNN	18	layer = 3	2	0.082	**0.004**

from being trapped at locally sub-optimal results. The result in Table 6 shows the effectiveness of our DMAV-aware gate-fusion algorithm.

To clarify the computational cost of our gate-fusion algorithm, we report its runtime overhead in Table 7. For three representative benchmark circuits, the absolute gate-fusion runtime ranges from 3.07 ms to 50.4 ms, corresponding to only 1.19%–5.05% of the total simulation time. These results show that, although gate fusion is not part of the core FlatDD workflow, it is a highly cost-effective optimization.

4.8 Comparison with MPS Simulation

To better understand the advantages and limitations of FlatDD relative to tensor network-based methods, we additionally compare FlatDD-v2 with Qiskit Aer's MPS simulator on representative circuit families, including DNN, quantum supremacy, QAOA, VQE, KNN, GHZ, and Swap test circuits in Table 8. The table reports the number of qubits (n), circuit parameter, maximum MPS bond dimension, and runtime for each circuit.

In MPS simulation, the *bond dimension* measures the size of the tensor connections used to represent the correlations between the qubit partitions. A small bond dimension generally indicates that a circuit is well suited for MPS simulation, while large bond dimensions increase the cost of the MPS representation. Therefore, the maximum bond dimension provides a useful indicator of MPS suitability.

As shown in Table 8, the FlatDD-v2 and MPS are effective in different simulation scenarios. MPS performs better on circuits with small bond dimensions, such as VQE-HEA, GHZ, and KNN. For example, the 20-qubit VQE-HEA circuit with layer=32 has a maximum bond dimension of 8, where MPS is faster than FlatDD-v2. In contrast, FlatDD-v2 performs better when the MPS bond dimension becomes large. For the 20-qubit DNN circuit with depth=8, the maximum bond dimension reaches 1024, and for the 18-qubit quantum-supremacy circuit with depth=8, it reaches 512; in both cases, FlatDD-v2 outperforms MPS.

These results indicate that FlatDD is not intended to replace MPS-based simulation for all circuits. Rather, the two methods target different scenarios. MPS is preferable for circuits with low entanglement and small bond dimensions, while FlatDD-v2 is more effective for medium-scale general circuits where MPS bond dimensions grow significantly larger.

5 Conclusion

In this paper, we have introduced FlatDD, a parallel quantum circuit simulator that combines the strengths of DD-based and array-based simulators. FlatDD parallelizes the simulation workload at multiple levels and leverages caching to reuse historical results. To further enhance the simulation performance for deep circuits, FlatDD introduces a gate-fusion algorithm to reduce the computational cost. Evaluated on commonly used quantum circuits, FlatDD has demonstrated 54.35× speed-up and 1.91× memory reduction compared to state-of-the-art simulators. Inspired by our success of GPU-accelerated computing [14–20, 27–36, 41–54, 56–62, 67, 68, 71–74, 77–87, 89, 90, 98, 99, 102], our future work will enhance the simulation performance using GPU.

Acknowledgments

This work is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141, as well as the Research Grants Council of Hong Kong SAR (No. CUHK14207523). This work is also supported by ACCESS – AI Chip Center for Emerging Smart Systems (sponsored by InnoHK funding), the JC STEM Lab of Intelligent Design Automation (funded by the Hong Kong Jockey Club Charities Trust), Hong Kong SAR. We also acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 101001318) and the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus.

References

[1] [n. d.]. Cirq. https://quantumai.google/cirq
[2] [n. d.]. Multiply–accumulate operation. https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation
[3] [n. d.]. OpenMP. https://www.openmp.org/
[4] [n. d.]. Qiskit. https://qiskit.org/
[5] 2020. qsim. https://doi.org/10.5281/zenodo.4023103
[6] 2024. mqt-ddsim. https://github.com/cda-tum/mqt-ddsim
[7] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. Fastflow: High-Level and Efficient Streaming on Multicore. *Programming multi-core and many-core computing systems* (2017), 261–280.
[8] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
[9] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *PACT*. IEEE, 116–128.
[10] Bela Bauer, Sergey Bravyi, Mario Motta, and Garnet Kin-Lic Chan. 2020. Quantum algorithms for quantum chemistry and quantum materials science. *Chemical Reviews* 120, 22 (2020), 12685–12717.
[11] Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. 2020. Training deep quantum neural networks. *Nature communications* 11, 1 (2020), 808.
[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.
[13] Lukas Burgholzer and Robert Wille. 2020. Advanced equivalence checking for quantum circuits. *IEEE TCAD* 40, 9 (2020), 1810–1824.
[14] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE DAC*.
[15] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE HPEC*.
[16] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM HPDC*.

[17] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE DAC*.

[18] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *AMTE*.

[19] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM ICCAD*.

[20] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE IPDPS Workshop*.

[21] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.

[22] Vlad Gheorghiu. 2018. Quantum++: A modern C++ quantum computing library. *PloS one* 13, 12 (2018), e0208073.

[23] Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. 2002. Quantum cryptography. *Reviews of modern physics* 74, 1 (2002), 145.

[24] Thomas Grurl, Jürgen Fuß, Stefan Hillmich, Lukas Burgholzer, and Robert Wille. 2020. Arrays vs. decision diagrams: A case study on quantum circuit simulators. In *IEEE ISMVL*. 176–181.

[25] Thomas Grurl, Jürgen Fuß, and Robert Wille. 2022. Noise-aware quantum circuit simulation with decision diagrams. *TCAD* 42, 3 (2022), 860–873.

[26] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen. http://eigen.tuxfamily.org.

[27] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE DAC*.

[28] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE TCAD* (2023).

[29] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM ICCAD*.

[30] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM DAC*.

[31] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM DATE*.

[32] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM ICCAD*.

[33] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM DAC*.

[34] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM ICCAD*.

[35] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE TCAD* (2023).

[36] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM DATE*.

[37] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *SC*. IEEE, 866–874.

[38] Dylan Herman, Cody Googin, Xiaoyuan Liu, Yue Sun, Alexey Galda, Ilya Safro, Marco Pistoia, and Yuri Alexeev. 2023. Quantum computing for finance. *Nature Reviews Physics* 5, 8 (2023), 450–465.

[39] Stefan Hillmich, Igor L. Markov, and Robert Wille. 2020. Just Like the Real Thing: Fast Weak Simulation of Quantum Computation. In *DAC*. 1–6.

[40] Stefan Hillmich, Alwin Zulehner, and Robert Wille. 2020. Concurrency in DD-based quantum circuit simulation. In *IEEE ASP-DAC*. 115–120.

[41] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM ICCAD*.

[42] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE ProTools*.

[43] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE HPEC*.

[44] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE IPDPS*.

[45] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD* (2021).

[46] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE HPEC*.

[47] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE DAC*.

[48] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*.

[49] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS*.

[50] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE DAC*.

[51] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM ICCAD*.

[52] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[53] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test (DAT)* (2021).

[54] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE TPDS* (2022).

[55] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE TCAD* (2022).

[56] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE HIPS*.

[57] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM ICCAD*.

[58] Tsung-Wei Huang and Martin Wong. 2016. UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. *IEEE TCAD* (2016).

[59] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM DAC*.

[60] Tsung-Wei Huang, P.-C. Wu, and Martin Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM ICCAD*.

[61] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE/ACM ICCAD*.

[62] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM ISPD*.

[63] J Stuart Hunter. 1986. The exponentially weighted moving average. *Journal of quality technology* 18, 4 (1986), 203–210.

[64] Intel. [n. d.]. AVX2. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intrinsics-for-avx2.html

[65] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 79–94.

[66] Shui Jiang, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei Huang. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *Proceedings of the 53rd International Conference on Parallel Processing*. 388–399.

[67] Shui Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE HPEC*.

[68] Shui Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM ICPP*.

[69] Chenyang Jiao, Weihua Zhang, and Li Shen. 2023. Communication Optimizations for State-vector Quantum Simulator on CPU+ GPU Clusters. In *ICPP*. 203–212.

[70] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th international conference on partitioned global address space programming models*. 1–11.

[71] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE DAC*.

[72] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM ASPDAC*.

[73] T.-Y. Lai, Tsung-Wei Huang, , and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM ICCAD*.

[74] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE DAC*.

[75] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Heim, Martin Roetteler, and Sriram Krishnamoorthy. 2021. SV-Sim: scalable PGAS-based state vector simulation of quantum circuits. In *SC*. 1–14.

[76] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM TQC* 4, 2 (2023), 1–26.

[77] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*.

[78] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE HPEC*.

[79] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE ICPADS*.

[80] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM GLSVLSI*.

[81] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE HPEC*.

[82] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *Euro-Par*.
[83] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE TPDS* (2022).
[84] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *Euro-Par*.
[85] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM ICPP*.
[86] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE DAC*.
[87] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.
[88] Ji Liu, Luciano Bello, and Huiyang Zhou. 2021. Relaxed Peephole Optimization: A Novel Compiler Optimization for Quantum Circuits. In *CGO*. 301–314.
[89] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM ASPDAC*.
[90] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE ESPM2*.
[91] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. 2023. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In *ASPLOS*. 252–265.
[92] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A Thornton, and Rolf Drechsler. 2015. QMDDs: Efficient quantum function representation and manipulation. *IEEE TCAD* 35, 1 (2015), 86–99.
[93] qpp. 2024. https://github.com/softwareQinc/qpp
[94] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking software and design automation tools for quantum computing. *Quantum* 7 (2023), 1062.
[95] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. 2016. qHiPSTER: The quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195* (2016).
[96] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *ASE*. 692–704.
[97] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2019. Full-state quantum circuit simulation by using data compression. In *SC*. 1–24.
[98] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE HPEC*.
[99] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.
[100] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: hybrid partitioner based quantum circuit simulation system on GPU. In *ICS*. 443–454.
[101] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. 2020. Quantum computational advantage using photons. *Science* 370, 6523 (2020), 1460–1463.
[102] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM ASPDAC*.
[103] Alwin Zulehner, Stefan Hillmich, Igor L Markov, and Robert Wille. 2020. Approximation of quantum states using decision diagrams. In *ASP-DAC*. IEEE, 121–126.
[104] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to efficiently handle complex values? Implementing decision diagrams for quantum computing. In *IEEE ICCAD*. 1–7.
[105] Alwin Zulehner and Robert Wille. 2018. Advanced simulation of quantum computations. *IEEE TCAD* 38, 5 (2018), 848–859.
[106] Alwin Zulehner and Robert Wille. 2019. Matrix-vector vs. matrix-matrix multiplication: Potential in DD-based simulation of quantum computations. In *IEEE DATE*. 90–95.